HZ BOOKS
华章教育

计 算 机 科 学 丛 书

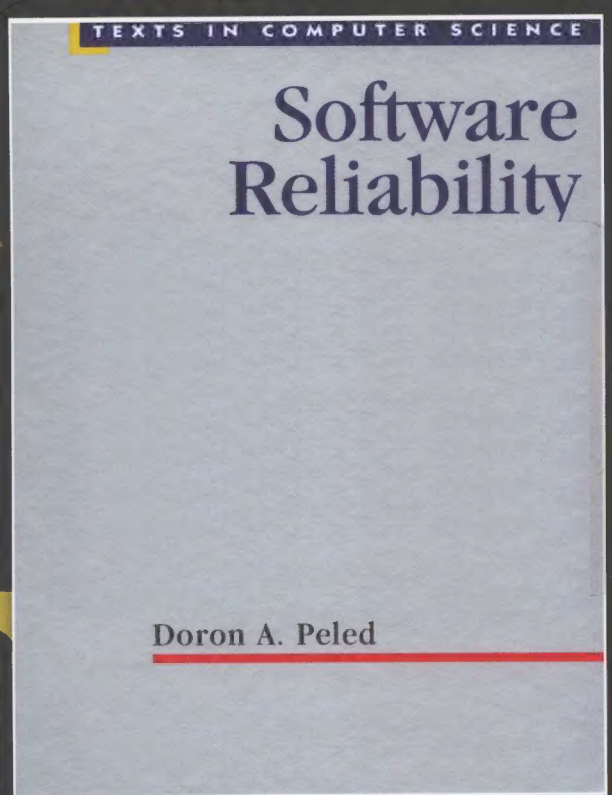
Springer

软件可靠性方法

（以）Doron A. Peled 著

王林章 卜磊 陈鑫 张天 赵建华 李宣东 译

Software Reliability Methods



机械工业出版社
China Machine Press

软件可靠性方法

Software Reliability Methods

用于创建可靠软件的形式化方法一直处于不断的开发和改进之中。最近，人们对于形式化方法工具的重要组成有了更深入的理解，从软硬件开发业界逐渐接受可靠性工具这一点就可以体现出来。

本书介绍了各种能解决软件可靠性问题的方法。理想情况下，形式化方法应该用起来直观，学起来简洁、快速，对开发过程的影响微乎其微。本书对各种方法进行了比较，揭示了它们各自的优点和缺点，同时紧扣自动机理论和逻辑这两个主题。在尽可能减少背景知识介绍的前提下，本书向非专家读者描述了多种技术，并且针对软件工程领域的研究人员和专业人士介绍了一些高级技术。

本书特点

- 集中介绍目前常用的重要软件可靠性方法，并将它们互作比较，这些方法包括：演绎验证、自动验证、测试和进程代数
- 为具体项目的软件选择过程提供有用信息
- 提供了大量的练习、项目和连续性的实例，方便读者学习形式化方法并能够亲手使用这些工具
- 介绍了支持形式化方法的数学原理
- 对于该领域未来的研究方向，以及开发新方法和改进现有技术提出了有益的见解

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com
华章网站 http://www.hzbook.com
网上购书：www.china-pub.com
封面设计：锡 彬

上架指导：计算机 软件工程
ISBN 978-7-111-36553-2
9 787111 365532
定价：45.00元

计 算 机 科 学 丛 书

软件可靠性方法

(以) Doron A. Peled 著

王林章 卜磊 陈鑫 张天 赵建华 李宣东 译

Software Reliability Methods

机械工业出版社
China Machine Press

本书通过大量的形式化表示和技术，向读者提供了各种用于提高软件可靠性的形式化方法，包括演绎验证、自动验证、测试以及进程代数。书中紧紧围绕逻辑和自动机理论这条主线，比较了各种方法的不同之处，并讨论了它们的优缺点。

书中包含一些在多个章节中使用的、具有连续性的实例，有利于读者通过跟踪这些实例来了解不同形式化方法的优缺点。本书还包括大量的练习和项目，可以使用软件可靠性工具来完成。

本书适用于从事软件开发的广大读者，尤其适合作为高年级本科生和硕士生的教材和参考书。

本书版权登记号：图字：01-2011-3360

图书在版编目（CIP）数据

软件可靠性方法 /（以）佩莱得（Peled, D. A.）著；王林章等译. —北京：机械工业出版社，2012.2

（计算机科学丛书）

书名原文：Software Reliability Methods

ISBN 978-7-111-36553-2

Ⅰ. 软…　Ⅱ. ①佩…　②王…　Ⅲ. 软件可靠性　Ⅳ. TP311.5

中国版本图书馆 CIP 数据核字（2011）第 242836 号

机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码　100037）

责任编辑：夏平

北京瑞德印刷有限公司印刷

2012 年 3 月第 1 版第 1 次印刷

185mm×260mm · 13.25 印张

标准书号：ISBN 978-7-111-36553-2

定价：45.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88378991；88361066

购书热线：（010）68326294；88379649；68995259

投稿热线：（010）88379604

读者信箱：hzjsj@hzbook.com

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自 1998 年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与 Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage 等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出 Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson 等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www. hzbook. com
电子邮件：hzjsj@hzbook. com
联系电话：(010) 88379604
联系地址：北京市西城区百万庄南街 1 号
邮政编码：100037

华章教育
华章科技图书出版中心

中文版序

Software Reliability Methods

很高兴看到我的著作《Software Reliability Methods》被翻译成中文。随着软件工业和硬件工业的兴起，研究提高计算机系统质量的方法日趋成熟，并逐渐成为较大的研究领域。形式化方法已经在蓬勃发展了，这不仅表现在大量的国际会议和致力于这方面研究的大学教授上，还体现在大公司里实际采用的工具和开发小组中。甚至有人声称，仅是投在验证软件（比如，软件测试）上的精力，就与编写软件的投入旗鼓相当。

程序员排错与软件开发的历史一样长。而如今，软件规模庞大，参与的开发人员众多，不再允许把检测和改正代码的责任加在程序员自己身上。同时，在人们日常生活中软件的广泛使用使得保持其正确责任重大。因此，软件测试和验证往往由独立专家小组使用能将大部分工作自动化的工具来执行。

本书着重论述在质量保证任务中所用到的主要方法，其中包括推理验证、测试和检验，此外还讨论如何协调这些方法以及它们之间的联系。在20世纪60年代后期，有学者提出用推理验证的方法来证明程序的正确性，用基于逻辑的数学形式化方法描述的程序来执行验证，并开发了证明系统来形式地证明程序行为与规约一致。开发这种证明系统时，必须研究程序的形式语义，将代码与程序行为及相关的形式化逻辑联系起来。早期的证明系统仅处理顺序程序，之后出现了能处理并行程序的证明系统。20世纪70年代后期，出现了最早的具有形式化定义的软件测试方法，并有了理论基础，此后出现了基于数学规律的代码采样和检查过程，因此能够用各种方法提高测试的效率，加强测试的效果。20世纪80年代初期，有人提出了验证有限状态系统的自动化方法，即模型检验，随后出现了各种相关支撑工具。

这三种软件可靠性技术在许多方面都大相径庭。对于推理验证技术来说，只有能够使用证明系统的逻辑学家才能驾驭。同时，由于其整个过程基本上都是细致繁琐的人力劳动，因此很难将其扩展应用于大规模甚至中等规模的代码。测试是现阶段保证软件可靠性的最佳方法。这种技术往往不对代码进行通彻的全面检查，也不保证能找到所有的编码错误或者设计问题。但是，它可以很好地用于大系统，如果仅从一种技术所能找到的错误以及耗用的投资来看，它的投入产出比最高。模型检验使用高端计算机、借助于高效算法和数据结构的好的研究成果，其最终目的就是自动完成全面检验，一方面其检验的通彻程度能与推理验证技术相媲美，另一方面其在大系统上的运用效果能和软件测试相匹敌。

最近十年，形式化方法研究上进行了巨大投入，其结果是利用现有的工具和技术，我们可以用更快、更节省内存的算法来检查更大规模、更多类型的系统。部分研究致力于发现效率更高的算法。另一部分研究的目标是将不同方法和技术组合在一起，利用各个方法的优点，减少其限制的影响，完成追求软件可靠性的整体任务。结合了验证的思路之后，测试工具就可以自动找到很好覆盖被检系统的测试用例集；组合了测试和模型检验的原理，允许我们在缺乏完整的设计描述的情况下直接测试系统。而用组合、抽象的技术，我们可以先对代码的紧缩版本或子集进行模型检验，然后直接利用所得到的结果或者将其与其他结果组合起来，得到对真正的更大的系统的推出结论。

形式化方法的研究牵涉了数学、计算机科学中的不同领域。本书中描述的主要技术是基于自动机理论和逻辑。而形式化方法、技术和工具中还用到一些更深的思想，比如机器学习、微分方程、数据结构、博弈论等。经过对其多年的研究，我已经确信这是最有意思的研究领域。同时，我也有幸见证了它从仅仅理论研究发展到被大的软硬件公司部署使用。但是，为了追寻更高效、更好用的技术，还需要克服很多困难，解决很多挑战。比如说，如何自动根据形式规约生成正确的代码？尽管已经有些很不错的结果了，但这一问题的研究仍处于起步阶段。

对这个令人兴奋而又实用的跨学科研究领域，我充满了热忱，并将其灌注在本书行文当中，现在它被翻译为中文并出版，作为作者，我最大的希望就是它能够感染、打动并帮助那些渴望学习、使用甚至研制软件可靠性方法的中文读者。

Doron A. Peled

2012 年 2 月 1 日

译者序

Software Reliability Methods

进入21世纪以来，“高可信软件”先后被美国、欧盟列为优先资助的研究方向，我国科技部和国家自然科学基金委员会也在2007年分别为其设立了重点项目和重大研究计划，由此广泛推动了国内高校和科研单位在相关领域的研究工作。译者所在的南京大学软件工程组长期以来一直在开展与可信软件方法和技术相关的教学和研究工作，Doron A. Peled的《Software Reliability Methods》近年来一直被我们用做研究生的入门教材，是进入我们研究组的研究生必读的书籍。该书围绕软件可靠性较为全面、系统地介绍了相关方法和技术，深入浅出、通俗易懂。该书还受到了图灵奖获得者E. Clarke的大力推荐，作者本人在相关领域也很有建树。

2010年12月底，我与来访的机械工业出版社编辑姚蕾女士交流时，她了解到我们在用这本书，建议我们将它翻译成中文，以便让更多的国内读者能够深入理解作者的观点以及书中介绍的方法，从而在研究与实践中放手尝试。我们接受了她的建议，从2011年1月初开始着手翻译。参与本书翻译工作的主要是南京大学计算机软件新技术国家重点实验室、计算机科学与技术系软件工程组的老师和学生，分工情况如下：王林章（第9，10章、结束语）、卜磊（第5，6章）、陈鑫（第7，8章）、张天（第4，11章）、赵建华（第1，2，3章），王林章、李宣东对全书译稿进行了统稿、审校和修改。在翻译过程中得到研究生崔展齐、陈华杰、丁文旭、姜鹏、李游、李袁奎、柳溪、潘敏学、陶永晶、王寒非、邢雨辰、叶楠、张琦、周筱羽等的帮助，在此对他（她）们的辛勤劳动表示感谢。也感谢编辑王春华女士、夏平女士为本书所做的大量工作以及耐心等待。

尽管本书的翻译工作前后历时一年，但这是在大家繁忙的教学、科研以及各项服务工作之余完成的，所以仍感到时间紧张，有些内容的翻译表达仍不够理想。我们采用自己的研究方法来理解原书内容，对原文中我们认为表达错误或不确切的地方，也通过注释进行说明。限于我们的水平，中文表达难免有不当之处，在此敬请读者批评指正。如发现文字错误，请发邮件给王林章（lzwang@nju.edu.cn）。

王林章

2011年12月31日

我很荣幸能够为 Doron Peled 关于软件可靠性方法的新书写序。当我第一次翻开这本书时，我立刻被这本书的覆盖范围之广所深深打动，它覆盖了下列方面：

- *规约和建模*
- 演绎验证
- 模型检验
- 进程代数
- 程序测试
- 状态与消息序列图

除了对每个方法进行了相当深入的介绍以外，本书还讨论了应当在何时选取何种方法以及在选择这些方法时所必须做出的权衡。书中结合当前最新的工具，使用很多具有挑战性的实例来说明各种技术。书中甚至告知读者在网络上到哪里去获取这些工具！我还没看见过其他任何覆盖同样内容的书籍能达到如此的深度。

同时，本书描述了应用形式化方法的过程：从建模和规约开始，然后选择一个合适的验证技术，最后测试程序。这些知识在实践中是十分必要的，但是却很少在软件工程的课本里面出现。绝大多数相关书籍都重点关注某一项技术，例如程序测试，而没有覆盖到其他的验证技术，或者对如何将相关技术结合在一起使用进行讨论。由于 Doron 已经为书中所描述的许多验证方法的发展做出了重要的贡献，因此他对这些关键问题的相关见解非常重要。

这本书适用于参与软件开发的广大读者，尤其适合高年级本科生软件可靠性课程或者硕士阶段软件工程课程使用。实际上，书中非常充分地标注了相关进阶文献，这些文献可以用做从事代码验证的软件工程师或者形式化方法方面研究人员的参考书目。

我刚刚和 Doron 一起合作完成了一本关于模型检验技术的书，在此过程中，他作为计算机科学家的智慧与作为作家的写作技巧给我留下了深刻的印象。我确信这本书将会取得巨大的成功。我向所有对软件可靠性问题感兴趣的读者强烈推荐这本书。

Edmund M. Clarke

前 言

Software Reliability Methods

许多书籍描述了如何通过应用形式化方法来提高软件的质量。然而，大多数书籍都是围绕着某个特定的方法，并将该方法作为软件可靠性问题的推荐解决方案。本书通过大量的形式化符号表示法和技术，向读者呈现了形式化方法更加全面的描述。书中将会比较它们之间的不同之处，并讨论它们的优点与缺点。

形式化方法的主要挑战之一在于怎样将研究人员开发出的技术传播到软件开发社区中去。最近一段时间，我们似乎开始对形式化方法工具的重要组成部分有了更好的理解。这表现为此类工具越来越多地被软、硬件开发业界所接受。理想情况下，形式化方法必须可以被直观地使用（最好是使用图形化界面），不要求使用者花费大量的学习时间，并且在开发过程中只产生比较小的额外开销。相比于 10 年或 20 年前，形式化方法更容易被人们所接受，这在硬件领域尤为显著。然而，不同的方法之间仍然存在激烈的争论。

本书重点是通过一系列技术来讲述形式化方法的主要原理。有很多在撰写本书时已有的先进技术没有包括在本书中。这些先进的技术涉及实时系统和混成系统、描述规约的形式化体系，以及一些专用的数据结构，比如二元决策图。本书没有包含这些特定的技术，但这并不意味着本书中的方法优于它们，而是因为本书描述的算法和方法已经应用于目前最先进的软件可靠性工具中了。挑选出来这些方法仅仅是想以举一反三的方式来表现形式化方法这一主题。然而，挑选时总是不可避免地倾向于选择和我的研究方向比较接近的主题。贯穿本书的主线是*逻辑和自动机理论*。有兴趣的读者可以在相关章节后列出的其他书籍和研究论文中找到这些高级方法的细节。

如果没有直接使用相关工具的实际经验，对形式化方法的学习就是不完整的。本书包含了大量的练习和项目，可以使用软件可靠性工具来完成它们。书中还有一些在多个章节中使用的、具有连续性的实例。学习形式化方法并且了解其优点和缺陷的一个有效方法是跟踪理解这些跨章节的实例。在一些情况下，后面的章节会详述某个在前面章节的练习部分提到的连续性实例。这样做的另一个目的是帮助读者检查对前面章节练习的解答（而不是仅仅提供一个明确的答案）。我们鼓励读者去检查所获得的关于这些例子的直觉感受可否帮助改进他们对之前练习的解答。

书中提供的绝大部分练习和项目可以选择某个工具来完成。尽管一些软件可靠性工具要求不菲的许可费用，但是很多工具允许出于非盈利目的免费使用。使用这些工具通常需要从这些工具的万维网页下载并按照网页上的指示进行安装。相关章节后面列出了一些工具和相应网页。需要注意的是，即使有些工具不需要取得许可证就可以使用，它们也常常要求使用者向工具的开发者发送一份信件，同意遵守工具的使用条款。在许多情况下，这些条款限定工具只能用于学术目的，并且开发者不对因使用工具而可能造成的损害负责。由于网页和网址往往会变更，同时也因为新的工具不断被开发出来替代已有的工具，因此我不能保证书中列出的网页信息一直有效。此外，本书不能够保证这些工具可以在任何环境下都正常工作。

不同的团体对形式化方法有不同的兴趣，当然不可能写出一本对项目经理、软件开发人员、质量保证团队和研究人员具有相同吸引力的书，但是我仍然尝试着在书中加入令每个读者群中的成员都感兴趣的内容。因此，读者在阅读的时候可以跳过那些理论性或技术性太强的章节。应当指出的是，本书的重点主要是*技术*，而不是*方法学*。

本书在描述一些形式化方法的同时还给出了相应的算法，理解这些算法对于运用这些方法

并不是必不可少的，但是理解它们可以更加深入地了解这些方法的工作原理。本书省略了大多数和书中的形式化方法相关的数学证明，但有时会包含简略的证明，以便增加读者的直观理解。

作者在此感谢下面这些参与和本书相关的启发性讨论并提出有益建议的人员：Nina Amla、Christel Baier、David Basin、Shai Ben-David、Roderick Bloem、Glenn Bruns、Ed Clarke、Dennis Dams、Xiaoqun Du、Kousha Etessami、Amy Felty、Elsa Gunter、Doug Howe、Orna Kupferman、Bart Knaack、Bob Kurshan、Bengt Jonsson、Leonid Libkin、Anca Muscholl、Kedar Namjoshi、Wojciech Penczek、Kavita Ravi、Natarajan Shankar、Natasha Sharygina、Marian Srenby、Richard Tefler、Wolfgang Thomas、Moshe Vardi、Igor Walukiewicz、Thomas Wilke、Mihalis Yannakakis 和 Lenore Zuck。事实上，写这本书的最大收获之一是有机会从实践者和专家就某些特定主题给出的建议和评论中进一步学习。

我并不是第一个引用刘易斯·卡洛尔（Lewis Carroll）的探险小说[⊖]中的词句的。但是，鲜为人知的是 Charles Lutwidge Dodgson（笔名是 Lewis Carroll）是一个研究逻辑可视化表示的数学家。他的“biliteral”（两字母）和“triliteral”（三字母）图是卡诺图的前身。卡诺图能够用一种易教易学的方式来表示逻辑，这也是很多形式化方法的最新研究趋势。

Doron Peled

2001 年 3 月于新泽西州莫雷山

⊖ 本书中所引《Alice's Adventures in Wonderland》中词句的中文翻译引自陈复庵先生所翻译的《阿丽思漫游奇境记》（中国对外翻译出版公司 1987 年版），所引《Through the Looking Glass》中词句的中文翻译引自许季鸿先生所翻译的《艾丽丝镜中奇遇记》（文化艺术出版社 1986 年版）。本书译者将书中主人公名译成“爱丽丝”。 ——译者注

目 录

Software Reliability Methods

第1章
Software Reliability Methods

引　言

“请示陛下，我从哪儿念起?”他问道。“从一开头念起，”国王非常庄重地说，“一直念到末尾，然后停止。”

刘易斯·卡洛尔《爱丽丝漫游奇境记》

在1999年年底，整个世界忧虑地等待着2000年的到来，人们忧虑的重点是控制要害系统的计算机可能会造成一些潜在的破坏。日历中年份的变化和计算机存储器中表示20世纪年份的传统方式是可能导致这些破坏的原因：传统的方法仅使用从00到99的两位有效数字来表示年份。这个出乎意料的小细节使一些人预感到将会发生的极大损失。它会影响到由软件驱动的电子系统，例如交通控制、核导弹、核反应堆、银行系统、抚恤金计划、电力和用水供应。仅美国就花费了超过1 000亿美元以应对这种危害，这就是“千年虫”。在日期发生变化的前一刻，一些人躲进了自制的避难所，同时手电筒和瓶装水成为普遍的需求。美国和俄罗斯的军事联合小组则在北美航空航天防御司令部（NORAD）度过了1999年12月31日的夜晚，他们共同监测天空，随时对可能发生的导弹计算机错误进行预警，这类错误可能会导致导弹失控而自动发射。午夜，1999年12月31日过去了，时间跨进了新的千年，这期间除了一些小故障以外，没有发生重大的事件。

计算机系统已经控制了我们生活中的方方面面，电话系统、商店的结账登记系统、票务预订系统、医疗系统、金融系统都已经高度计算机化。大多数情况下，计算机之间的金融数据通信已经代替了实际的纸币交易。计算机甚至可以负责处理民用飞机飞行中要求的很多活动。计算机系统的故障已经导致了一些严重的后果，其中包括死亡事故、自动关闭要害系统以及金钱损失。

软件开发行业在过去的数十年里正以史无前例的速度发展。硬件成本，尤其是内存价格不断下降。因特网实际上已经将整个世界变成了一个巨大的计算机网络，在这个网络里，通信、信息处理和商业交易持续不断地在线进行着。随着这些技术的不断更新，软件开发行业也在发生巨变。现在，由一个天才程序员单枪匹马地开发出整个软件系统的情况已经不再可能，取而代之的是，数十个甚至数千个程序员一起参与到开发过程中。不同的程序员开发同一个程序的不同模块，生产出几千行甚至数百万行代码。这些程序员可能在不同的地方工作，甚至可能素昧平生。

毫不奇怪，软件开发过程中的质量控制变得越来越困难，确保不同软件开发团队开发的产品能够成功地组合在一起正确运行则是一项不简单的任务。应对这一问题的主要技术之一是运用适当的设计方法学，软件工程学因此提出了很多技术来提高软件产品的质量。

正如许多软件工程方法所指出的，即使遵循了很多优秀设计准则和优良编程规范，程序可能仍然包含错误。不同的统计表明，即使由富有经验并训练有素的程序员来编写代码，每千行代码中总是会包含少量错误。因此，使用一些方法来消除程序代码中的人为错误就变得非常重要，这就是形式化方法技术（formal methods technique）的目的所在。多种软件工程方法试图指导软件的开发过程，但形式化方法的目的是为开发过程提供一些辅助性的技术和工具，用于发现并指出软件中的潜在问题。

1.1 形式化方法

形式化方法是一系列用于描述和分析系统的符号表示法和技术。这些方法被称为形式化方法的原因在于它们是以一些数学理论为基础的，例如逻辑、自动机和图论，它们都致力于提高系统的质量。形式化规约技术引入了一种能够准确、无二义地描述系统属性的方法。它对于消除误解十分有效，也可以进一步用于系统调试。形式化分析技术能够用于验证系统是否满足它的规约，或者系统化地发现它不能满足规约的情况。形式化方法能够减少查找设计或编码错误所需要的时间。它还能够有效降低因为未能在系统部署之前发现错误而导致损害的风险。本书中我们将会关注用于软件的形式化方法，我们也称之为*软件可靠性方法*（software reliability method）。

这些技术有时由程序员自己使用，但是通常情况下，使用它们是专门的软件质量保证团队的责任。在后一种情况下，开发人员和那些关注可靠性的人员（例如测试团队）分工合作，以实现更为可靠的软件为目标，一起创建有利的协同工作方式。实际上，在许多情况下，质量保证团队的规模要大于开发团队的规模。

形式化方法研究的早期，研究的焦点是保证系统的*正确性*。这里所说的系统正确性是指系统能够满足客户提供的规约。*演绎软件验证*（deductive software verification）技术是当时主要的形式化方法技术研究之一。这一领域的先驱，例如 Dijkstra、Floyd、Gries、Hoare、Lamport、Manna、Owicki、Pnueli 等，提出了不同的可用于验证程序正确性的证明系统。这类技术的目标是形式化证明一些关键系统，例如导弹、航空控制。有人建议验证方法也能够用于辅助软硬件系统的开发 [94]，根据这一建议，程序开发时从它们的形式化规约开始，按照逐步精化的方式最终得到实际的代码，这个过程中的每个精化步骤都保持了正确性。

人们很快认识到了形式化方法的一些局限性。软件验证方法不能够保证*实际*代码的正确性，而是允许人们对实际代码的一些*抽象模型*进行验证。因此，由于真实系统和其抽象模型可能有所区别，所以对模型的正确性证明可能对实际代码无效。不仅如此，正确性证明本身可能不正确。证明的过程通常只覆盖了系统功能的一小部分。其中的原因在于验证是针对一个给定的系统规约而进行的，而这个规约是手工定义的，有些时候它可能不完整，忽略了系统的一些重要属性。而事实上，评定一个给定的规约是否完整通常是很困难的，有时甚至不可能完成。尽管演绎验证的某些部分可以自动完成，但是一般而言它仍是一种靠手工完成的技术。因此，演绎验证主要通过很小的例子进行展示，而且需要大量的时间和专门的技术。

尽管演绎软件验证有这些局限性，但它还是取得了一些重要的成就，它通过定义*不变式*这个概念影响了软件开发过程。不变式是一个正确性断言，在执行过程中，它必须在某些特定的控制点上成立。它断言了程序中变量之间的某些联系。程序员向程序代码中添加不变式可以增加他们对该程序的运行方式的直观理解。不仅如此，他们还能够插入简单的运行时检查代码，以检查不变式在特定的程序控制点上真的成立。如果某个不变式不成立，程序的正常执行就会被中断，同时给出一个警告信息。

最近，新的并且更高效的工具正不断地开发出来，为演绎软件验证提供更好的支持。这些工具试图将部分证明过程机械化。净室方法 [98，99]（本书的后面会讲述该方法）是一个非形式化地应用软件验证的例子，它与其他软件可靠性方法一起被用于一个软件开发方法论中，致力于提供高质量的软件。

早期的*自动化验证技术*是由 West 和 Zafiropulo [146，150] 提出的。这些方法可以用于各种*有限状态系统*（finite state system），例如硬件电路和通信协议。美国的 Clarke、Emerson [28，42] 和法国的 Quielle、Sifakis [120] 首先提出了*模型检验*（model checking）的概念。该技术可以验证一个有限状态系统相对于给定的时序规约的正确性。从较小规模的例子（如交通信号灯

控制器）开始，这些方法逐渐成熟并形成了数百万美元的工具产业。

新的自动化和半自动化的验证技术表明了自动验证方法的可扩展性。这些技术包括二元决策图 [24]、偏序约减技术 [55，113，142]、对称约减 [43]、归纳技术 [81，149]、抽象技术等。这些技术证明了验证技术不仅仅局限于理论中，在实践中也一样出色。但是，在使用自动验证时仍然需要考虑到若干限制，这些限制包括处理较复杂的数据结构（如队列和树）时效率低下。此外，自动验证方法通常局限于有限状态系统，因而它们最适用于验证通信协议和某些算法的抽象表示。它们可能无法处理那些带有实数与整数变量或带有指针与数组引用的成熟程序。和演绎验证一样，自动验证技术也需要首先对系统进行建模，而不是直接进行验证。这样可能会造成被验证的模型对象与实际系统的不一致。

软件测试（software testing）也许是使用最为频繁的质量保证方法了。它不会提供对系统的全面检查，软件测试的重点是根据某些覆盖准则对程序运行过程进行采样，并将程序的实际行为与程序规约中规定的期望行为进行比较。测试主要是根据实验证据和经验来完成的，并且常常通过非形式化的方式完成。测试方法的优点在于能够对一个实际的系统，而不是系统的模型，进行检查。和模型检验不同，测试并不局限于有限状态系统。但是测试不是和演绎验证、模型检验一样的全面性技术，它不能覆盖所有可能的执行过程。因而即使是经过彻底测试的代码，其中仍可能包含错误。

一些形式化方法术语

不同形式化方法的研究者和实践者以不同的方式使用验证（verification）这一术语。有些时候，验证仅仅是指获取系统的形式化正确性证明的过程，也就是说狭义上的验证即是演绎验证。在其他语境下，验证是指试图寻找程序中错误的任何活动，包括模型检验和测试。

因为本书中讲述了不同的形式化方法技术，我们有时候不得不为某些术语选择适当的解释，而这些术语在不同团体中的解释可能并不一致。在提到验证时，我们指的是某个手工或自动化技术的应用过程，这些技术可以确定代码是否满足某个属性，或者它的行为是否和某个高层次描述相符。根据这个定义，我们主要指的是演绎验证和模型检验这两类活动，但不是指测试，因为后者更像是抽样检验，而不是一个全面的正确性检查。

我们会区分下面两种不同的活动：验证一个软件是否符合特定的规约属性（property）（有时也称作确认（validation）[70]），以及验证对软件的两个描述之间的一致性（conformance）。在后一种情况下，其中一个描述通常包含更少的细节，即更加抽象。

1.2 开发与学习形式化方法

形式化方法包括规约、验证和测试技术，这些方法被用于提高软件开发和硬件设计的质量。在过去的 20 多年中，人们认识到了各种方法的好处、优势、权衡和局限性。理解形式化方法的局限性和介绍它们的成功案例同样重要。例如，经验表明计算方法比那些需要大量人工技巧的方法更好。从另一方面看，自动化方法的处理范围也有局限性，例如它们常常只能处理中等规模的有限状态系统。作为形式化方法研究成熟的表现之一，人们总是用怀疑的目光看待那些试图保证系统正确性的方法，而倾向于使用那些以寻找错误为目的的方法。

各种形式化方法的工具和支撑系统首先由大学和研究所的研究者倡导。最近几年中，我们已经目睹一些公司开发出他们自己的形式化方法和相应的工具，尤其是那些通信和硬件领域的企业，因为其产品的可靠性是一个关键指标。在其他地方，人们开发了一些接口工具，将软件转换成现有工具可以接受的形式，这些工具的输出可能又被转换回某些规定格式。我们也开始逐渐看到一些现有的工具可以增强软件的可靠性，但是这些工具中的大部分都和软件测试或软件

建模中的不同任务相关。

形式化方法的研究还是相对较新的，选择正确的技术和工具是一件困难的事情，而开发一种新的验证或测试工具则需要相当大的工作量。本书中，我们将概述各种形式化方法之间的异同，同时也指出它们可以怎样组合在一起，弥补各自的不足。

当一个新的软件产品被开发出来时，需要决定是否使用某个特定的软件可靠性工具或相应的技术。在有些情况下，应该尝试着开发一个新的工具或裁剪现有的工具以满足某些特定的需要。软件开发中担负不同角色的人们有着不同的考虑。下面列出了一些相关的问题：

- 项目经理
  - —在我的软件项目中使用形式化方法会获得什么好处？
  - —需要哪些时间、人力和金钱上的投入？
  - —我是不是应该建立一个内部的形式化方法小组，以便开发新的符合我部门需求的工具？
- 质量保证团队主管
  - —我们应该在开发过程的哪个阶段使用形式化方法？
  - —我如何合理安排使用形式化方法的活动和其他的开发活动？
  - —我们需要多少人力来完成这项任务？他们需要什么样的资质？
  - —我们应该如何雇用或训练员工来应用形式化方法？
- 工程师（用户）
  - —哪个工具或技术能够最好地帮助我们获得更高的可靠性？
  - —它的成本是多少？
  - —这个工具能够提供什么样的支持？
  - —当前正在开发系统的最适当的形式化表示是什么？
  - —使用不同的形式化方法找到错误的概率各是多少？
  - —什么样的形式化规约最适合被开发的系统？
  - —为达到我部门的目标，开发一个新方法或改进一个已有方法需要多长时间？
  - —我们是否需要特殊的设备或软件来支持这些工具？
  - —我们怎样使用选定的某个方法或工具对被开发系统（或其一部分）进行建模？
  - —我们应该如何解释形式化方法给出的结果？例如，当工具报告错误时，系统真的有错吗？当没有发现错误时，我们对系统的把握有多大？
  - —当我们使用的工具失败了怎么办？例如当验证工具没有在适当的时间内得出结果或者出现了内存耗尽的情况。
  - —我们的规约正确吗？其中是否包括了一些自相矛盾的情况？它们是不是包括了该系统所有必要的需求？
- 形式化方法研究人员
  - —我怎样才能增加形式化方法技术的*表达能力*（expressiveness），以便可以规约和验证更多的系统需求？
  - —我怎样才能提高这些技术的*效率*（efficiency），使得这些方法和工具能够运行得更快并能够处理规模更大的实例？
  - —是否存在*启发式的方法*（heuristics）能够在很多实际情况下比标准方法更好？
  - —我怎样向潜在的用户推广新技术？
  - —软件开发人员常使用哪些表示方法？我怎样将它们整合到我开发的技术和工具中去？

我们无法在书中回答所有这些问题，因为对其中某些问题的回答强烈依赖于被开发的特定系统、环境和开发组织。但是我们试图通过提供一些现代形式化方法的相关信息、它们的用途和

局限性来解答这些问题。我们的目的是讲述主要的方法，并说明它们的长处和弱点。这将会使潜在的用户获得更多有关它们的能力与局限性的知识，以便他们可以为自己的目标选择适当的方法，并快速有效地加以使用。

1.3 使用形式化方法

使用形式化方法可以达到多种目标：

- 在开发过程的不同阶段，获得关于一个系统共同的、形式化的描述。形式化规约（specification）允许参与开发的不同小组共同使用对系统或其属性的一致描述。这个描述有着清晰、无二义的含义。形式化规约向设计人员、编程人员、质量保证团队和此后的用户提供了一系列的文档，这些文档可以用于解决对系统预期行为的误解和争论。
- 研究形式化方法的目的是找出系统开发过程中引入的错误，它可用于帮助设计人员和开发人员发现、定位并分析错误，从而增加系统的可靠性。
- 可将形式化方法集成到开发过程中，并起到辅助作用。一些形式化方法工具不仅可以对系统的设计进行建模和描述，而且可以通过某种方式自动或半自动地把设计转换成初步实现。例如，描述系统的动态行为的工具可能会自动地产生出相应行为的代码。

形式化方法几乎能够用于软件开发项目的每一个阶段。人们能够使用形式化规约描述期望的系统特性和系统规约。测试和验证在这个早期阶段就可以开始进行。系统的初始设计就已经可以使用早期错误检测算法进行处理，这些初始设计甚至可能只是可行性研究中获得的一组可能场景。这些算法可以就一些潜在的问题对开发人员示警，例如功能之间或功能与环境之间的意外交互。

测试和验证可以用于开发的不同阶段。在开发过程中，错误发现得越早，它造成的损失也就越少，修复它的代价也就越小。然而，即使形式化方法已经广泛应用于早期开发阶段，新错误仍然会渗透到被开发系统中，因而在后面的阶段不断地使用形式化方法是一个有用的实践方法。

软件开发方法论规定了如何获得系统的需求，规划时间表，估算成本与人力资源，以及在不同的小组之间分配工作等。在这样的一个方法论的指导下，在一个项目中恰当地使用形式化方法就可以得到最高的效率。应该为增强被开发软件的可靠性保留专门的时间和人力资源。

在现实中，形式化方法的倡导者常常需要与其他开发活动相竞争。下面几个原因会导致这种情况发生：

- 形式化方法相对来说是新生事物，其中的一些方法在最近（20年前，甚至更短）才被提出。
- 人们倾向于尽快地完成开发并交付产品，各个开发活动的时间限制都很紧。从表面上来看，形式化方法似乎需要在时间、金钱和人力资源上投入更多。人们（特别是在尚未采用这种技术的机构里）常常忽略这样的事实：形式化方法能通过找到错误来节省时间和金钱。
- 形式化方法的研究经常致力于提高该技术的表达能力和效率，这通常会导致人们忽略人机界面的问题，而人机界面在吸引新用户方面是很关键的。由于形式化方法是基于对系统的数学分析，所以一些技术要求潜在用户自学怎样使用一个新的数学框架和符号系统。直到最近人们才渐渐接受这样一个事实：形式化方法的开发人员需要采用潜在用户已经使用的框架和表示符号。

许多软件开发人员和软件工程师对使用形式化方法仍然怀有疑虑。造成这种情况的主要原因之一可能是使用了一些像“验证”这样的术语，这些术语暗示了能够对被检查软件的某些绝对保证，但是这类绝对保证实际上是不可能实现的。为此，软件验证工具的许可证通常都会包含对检测被检软件中的部分或全部问题的免责声明，这看起来似乎是一个营销问题。通过解释使用形式化方法的优点和它们的局限性，我们可以更好地分析这个问题。

当前存在大量的对形式化方法的常见误解和偏见：

形式化方法只能由数学家使用。

形式化方法是基于数学理论的这一事实表明这些方法的研究人员很可能是数学家、计算机科学家或者电气工程师。一些方法，例如演绎验证，确实需要很高的数学技巧，但是许多现代的形式化方法已经设计得使软件工程师只需要很少的培训就可以使用。这是由于形式化方法的开发人员意识到了这样的事实：当他们的方法是基于目标用户的方法和工具，而不是要求用户学习新的形式化理论时，他们常常会获得成功。

使用形式化方法会拖慢项目的进度。

的确，使用形式化方法是会花费不少时间。在开发过程中增加使用形式化方法，需要安排一定数量的人力并花费一定时间参与到这个活动中来，但是经验表明使用形式化方法，尤其是在早期开发阶段使用，会减少系统调试和交付系统所需要的时间。在验证和测试系统上花费时间是值得的，因为顾客经常愿意为系统的可靠性支付更多的金钱。

验证过程本身就易于产生错误，这又是何必呢？

人们的确不能够对软件的正确性有完全的把握，在对软件建模时，以及为了适应验证和测试工具而对软件进行转换时都会产生错误。形式化方法工具也可能包含缺陷，并且人工验证很显然是容易出错的。事实上，这些问题并不仅仅存在于理论层面，当将形式化方法应用到实践中时也会产生这个问题。重点在于认识到形式化方法的目的并不是提供*绝对的*可靠性。因此，形式化方法的优势在于提高可靠性。统计研究表明，它们的确成功地实现了这一目标。

1.4 应用形式化方法

形式化方法能够用于软件开发过程的各个阶段，从早期设计到产品完成时的验收测试。理想情况下，这些形式化方法的使用会通过某种方法论集成到软件开发过程中，因此开发时间表应该包括进行测试和验证工作的明确时间段。确定软件已经通过特定的有效性检查可以被看做是软件开发过程中的里程碑。

遗憾的是，形式化方法的有效性随着被检测对象大小的增加而减小，这个问题有时被称做*状态空间爆炸*（state space explosion）。在本书中会详细地讨论这个问题。因此，将形式化方法应用到早期的设计和开发阶段更为有效，在这些阶段被检测的对象相对小一些。尽早使用形式化方法的另一个原因是早期阶段发现的错误造成很大破坏的可能性较小，并且也比较容易修复。

现今，软件系统由数千行到数百万行代码构成。为了应对软件开发的最后期限，软件开发通常由多个程序员小组共同进行，有时这些小组分布在不同的地方，因而将形式化方法应用于整个系统是很困难的。因此，现在的研究趋势是寻找可组合的方法，这种方法试图对代码的不同部分进行单独验证，然后做出对整个系统的论断。

仔细考虑一个理想化的例子，一个系统由两个子系统 A 和 B 组成。我们希望检验系统是否满足某个属性 p。在试图进行组合验证的时候，我们可能先尝试证明 A 满足某个属性 p_1，B 满足某个属性 p_2，那么我们希望证明：A 和 B 分别满足属性 p_1 和 p_2 则蕴涵整个系统满足属性 p。当子系统 A 和 B 是独立开发完成的时候，有一个团队检查 A 是否满足属性 p_1，另一个团队则检查 B 是否满足属性 p_2。这会存在多种可能性，例如允许开发 A 的团队检查 B，同时开发 B 的团队检查 A（让开发人员检查自己编写的代码通常不是一个好主意，第 9 章将会详细解释）。从另一个角度说，可组合的验证和测试增加了安排系统开发时间表的灵活性，因此组合的形式化方法非常值得。遗憾的是，正如我们后面将会看到的，有些情况很难实现可组合性。

形式化方法的应用对象并不仅限于软件系统的代码，人们可以一开始就检查需求中可能出现的矛盾。对于系统的早期设计，即使只包含少量的例子，形式化方法也可以验证其一致性。有

些软件开发方法，例如净室方法 [98，99]，在整个开发过程中都使用形式化方法来精化代码，试图使得软件在各个阶段都能够保持相对于前一阶段的正确性。理想情况下，人们能够从系统规约开始，然后反复对它进行精化并保持每个阶段的正确性，最后获得满足规约的代码。

本书中讲述的一些方法也可用于硬件的规约、验证和测试。尤其是，这些方法的某些变体包含了能够有效处理硬件的优化技术 [24]。随着软硬件设计的最新发展，同时包括硬件和软件组件的系统正在被开发出来。有时在一块芯片上实现某个关键的软件，而硬件设计现在是使用编程语言来实现的。随着硬件和软件之间的界限越来越模糊，并且开发软硬件模块相结合的系统具有不断增长的趋势，我们需要提供能够适应这二者的、有效的形式化方法 [80]。

应用形式化方法需要花费一定的成本，尽管采用更有效的方法有助于减少成本，但是没有办法完全消除这些成本。从另一方面来看，使用形式化方法可以消除一些由错误引起的不幸情况，因而从整体上来看可能会节省一些成本，在一些极端情况下甚至能够拯救人的生命。因此在某些情况下，应用形式化方法的投入等于甚至大于所有其他开发活动的总投入，这并不是不合理的。获取可靠性通常是一个很好的投资，有些时候甚至是无价的。

1.5 本书概要

形式化方法的研究已经产生了许多有趣而且实用的成果。一些基本的形式化方法已经显示出了一定的软件分析能力，目前的改进旨在达到更高的效率和自动化程度，以便将这些方法应用到更广泛的系统当中。在形式化方法已经成熟的领域，我们看到了基础技术衍生出的不同版本，某些版本进行了针对专门应用的优化。一本介绍性的书籍不可能将所有最新的研究成果都囊括在内。但是，对于很多基本方法和当前的热门方法，本书呈现的原理是相通的。

本书重点关注形式化方法的两个重要组成部分：*逻辑*和*自动机理论*。只要理解这些概念以及它们在对系统进行形式化分析中的作用，就能很容易地掌握这些基本思想以及这一领域新的发现。

我们首先介绍一些基本的数学背景知识（第 2 章）。逻辑（第 3 章）言简意赅地定义了一些语法，这些语法具有形式化语义，并且能够从已有事实推导出新的事实。它可以用于对系统及其属性的形式化描述。某些具体的逻辑，例如一阶逻辑（第 3 章）或者线性时序逻辑（第 5 章），经常被用于软件规约。相关的概念，例如进程代数（第 8 章），则经常被用于分析系统内部或者系统与环境之间的相互作用这一微妙问题。使用逻辑表示方法不仅仅限于提供一个精确和无二义性的描述。逻辑常常带有相应的证明系统，这样可以允许我们手动或自动地进行推理证明。这样的证明系统可以用于程序验证（第 7 章）。

另一个在形式化方法中扮演重要角色的数学理论是自动机理论（第 5、6 章）。一个自动机是一个数学结构，基于系统状态以及状态之间的迁移，它可以描述系统的不同行为。人们常常使用自动机的不同版本对软件系统进行建模。自动机理论是计算机科学的基础之一，因而许多数学工具可以用于自动机分析。形式化方法中使用这类工具的目的是对系统进行自动分析。

使用形式化方法时，通常首先对被检验系统进行数学建模（第 4 章）。被建模系统以及它的属性的形式化规约（第 5 章）需要通过某个选定的形式化描述方法来给出。然后人们就能够应用某些技术来获得对系统正确性的高度信心。主要的方法是模型检验（第 6 章），演绎验证（第 7 章）和测试（第 9 章）。人们也能够通过对一个设计进行逐步精化的方式来增强可靠性，这种精化过程中必须维持不同版本之间的一致性（第 8 章）。

本书的最后几章描述了一些更高级的技术。我们演示了怎样将不同形式化方法的优点结合在一起（第 10 章）。形式化方法中最近的成功趋势之一是使用可视的形式化描述方法（第 11 章）。最后，我们对本书中描述的方法，以及形式化方法研究中的一些新方向进行了讨论（“结束语”）。

第2章
Software Reliability Methods

预备知识

她说到这里，抬头一看，那只猫又坐在一根树枝上了。

刘易斯·卡洛尔《爱丽丝漫游奇境记》

软件可靠性方法建立在一些数学原理之上。通常这些方法的使用者并不一定要掌握相应的数学理论。本书中，我们将既展示这些技术本身，也展示一些构成这些技术的原理。因此，我们将必然会使用到一些数学术语。本章概括了本书后面将用到的一些概念和理论。如果读者之前对集合论、图论、复杂度理论和可计算性的基本概念已经有所了解，则可以跳过这一章。

2.1 集合表示法

一个集合就是将一组有限或无限多个元素聚集在一起。集合会忽略重复的元素。对于有限集合，它的元素可以在一对匹配的花括号之间罗列出来，比如 $\{1, 3, 5, 9\}$。集合的另一种表示方法是 $\{x|R(x)\}$，这里 R 是对 x 可取值范围的某种描述。例如，$\{x|even(x)$ 且 $x>0$ 且 $x<200\}$ 是 2 和 198 之间的偶数的集合。空集是一个特殊的集合，用 $\varnothing$ 表示，空集不包含任何元素。一个集合的大小是指它所包含的元素的个数。集合 A 的大小用 $|A|$ 表示。显然，$|\varnothing|=0$。

下面是一些能作用于集合的运算：

交运算 $A\cap B$ 表示既在 A 中又在 B 中的元素的集合。

并运算 $A\cup B$ 表示在 A 中或 B 中（或两者都在）的元素的集合。

差运算 $A\backslash B$ 表示在集合 A 中但不在 B 中的元素的集合。

幂运算 2^A 表示所有的可以由集合 A 中一些元素构成的集合的集合。要注意的是 2^A 是一个所有元素仍是集合的集合，这些元素都是 A 的子集。如果 A 有有限多个元素，那么 2^A 的大小就是 $2^{|A|}$。

例如，如果 $A=\{1, 3, 5, 6, 8\}$ 且 $B=\{3, 6, 9\}$，那么 $A\cap B=\{3, 6\}$，$A\cup B=\{1, 3, 5, 6, 8, 9\}$，$A\backslash B=\{1, 5, 8\}$。集合 B 的幂集，即 2^B 是

$$\{\varnothing,\{3\},\{6\},\{9\},\{3,6\},\{3,9\},\{6,9\},\{3,6,9\}\}$$

如果从上下文中能够确定所讨论元素来自一个给定的域 $\mathcal{D}$，例如，整数

$$Int = \{\cdots, -2, -1, 0, 1, 2, \cdots\}$$

或者自然数

$$Nat = \{0, 1, 2, \cdots\}$$

那么我们可以计算一个集合的补集。集合 A 的（相应于集合 $\mathcal{D}$ 的）补集，是包含了集合 $\mathcal{D}$ 中所有不在集合 A 中的元素的集合，用 $\overline{A}$ 表示。换句话说，$\overline{A}$ 就是 $\mathcal{D}\backslash A$。

我们用 $x\in A$ 来表示一个元素 x 属于一个集合 A。如果 x 不属于 A，我们写作 $x\notin A$，我们可以通过下列方式来比较一对集合：

- $A\subseteq B$，如果对于每一个元素 $x\in A$ 同样有 $x\in B$。我们说 A 包含于 B 或者 A 是 B 的一个子集。对于这种情况，我们也可以写作 $B\supseteq A$。
- $A=B$，如果 A 包含于 B 且 B 也包含于 A。我们说 A 和 B 是相等的。
- $A\subset B$，如果 A 包含于 B 但 A 不等于 B（也即 B 必须至少包含一个不在 A 中的元素）。我们说 A 真包含于 B 或者 A 是 B 的一个真子集。我们也可以写成 $B\supset A$。

例如，令 $B=\{1, 2, 3, 4, 5\}$，$A=\{2, 4, 5\}$。那么我们就可以说 $A\subseteq B$。事实上既然$A\neq B$，我们就有 $A\subset B$。

两个集合 A 和 B 的笛卡儿积，用 $A\times B$ 表示，是一个有序对的集合。在 $A\times B$ 中，每一个有序对的第一个元素来自 A，第二个元素来自 B。例如，如果 $A=\{q_0, q_1\}$，$B=\{s_2, s_3\}$，那么 $A\times B=\{\langle q_0, s_2\rangle, \langle q_0, s_3\rangle, \langle q_1, s_2\rangle, \langle q_1, s_3\rangle\}$。对于有限集合 A 和 B，如果 $|A|=m$ 且 $|B|=n$，那么 $|A\times B|=m\times n$。我们把两个元素列在尖括号（“〈”和“〉”）或圆括号（“(”和“)”）之间来表示一个有序对。通过反复应用笛卡儿积，我们可以获得更复杂的元素，例如三元组、四元组、六元组或者更通用的 n 元组。

一个 n 元关系是在某个域上的 n 元组的集合。每个 n 元组都是一个恰巧有 n 个有序元素的对象。关系的一个例子就是整数之间的“大于”关系，通常用“>”表示。这是一个二元关系，即元素对的关系。我们知道下列属于这个关系的元素对：

$$(3,5),(7,9),(3,11)$$

而下列的元素对则不属于这个关系

$$(3,3),(7,3),(0,-1)$$

一个关系还可以定义于多个域上，这样 n 元组中每个位置上的值都取自一个特定的域。

对于一个二元关系 $R\subseteq\mathcal{D}_1\times\mathcal{D}_2$，我们用 R^{-1} 表示关系 $\{\langle x, y\rangle\mid\langle y, x\rangle\in R\}$。要注意的是 $R^{-1}\subseteq\mathcal{D}_2\times\mathcal{D}_1$。有时我们说 R^{-1} 是 R 的反置。我们经常用 $R(x_1, \cdots, x_n)$ 来表示 $(x_1, \cdots, x_n)\in R$。对于二元关系，我们还使用中缀表示法，写作 x_1Rx_2。

一个二元关系 $R\subseteq\mathcal{D}\times\mathcal{D}$ 的传递闭包用 R^* 表示。传递闭包的定义如下：$(x, y)\in R^*$ 当有一个序列 $z_0, z_1, \cdots, z_n$ 满足对于 $0\leqslant i<n$ 都有 $(z_i, z_{i+1})\in R$，$z_0=x$ 且 $z_n=y$。

一个 n 元函数（或者说，映射）可以被视作一个包含 $(n+1)$ 元组的约束关系，每个元组的前面 n 个元素唯一确定了第 $(n+1)$ 个元素。也就是说，不可能有两个 $(n+1)$ 元组，它们的前 n 个元素相同，但第 $(n+1)$ 个元素不同。定义于域 $\mathcal{D}_1$，$\mathcal{D}_2$，$\cdots$，$\mathcal{D}_n$ 上，结果值来自于域 $\mathcal{D}_{n+1}$ 的函数 f 表示形式如下 $f:\mathcal{D}_1\times\mathcal{D}_2\times\cdots\times\mathcal{D}_n\rightarrow\mathcal{D}_{n+1}$。例如，整数加法，表示为“+”，是一个二元函数，即 $+:int\times int\rightarrow int$。可以将一个 n 元关系看做一个具有同样元数的函数，这个函数返回 TRUE 或 FALSE。这样，每个 n 元组被扩展为一个 $(n+1)$ 元组，其中最后一个布尔值元素是由前 n 个元素唯一确定的。要注意的是关系和函数符号经常会被“重载”，具有多个解释。例如，关系符号“>”既代表整数域上的“大于”，也作用于实数域之上。

等价关系 $\sim$ 是一个作用于某个域 $\mathcal{D}$ 上的满足下列条件的二元关系：

自反性　对于每个元素 $x\in\mathcal{D}$，$x\sim x$。

对称性　对于任意两个元素 x，$y\in\mathcal{D}$，如果 $x\sim y$，那么 $y\sim x$。

传递性　对于任意三个元素 x，y，$z\in\mathcal{D}$，如果 $x\sim y$ 且 $y\sim z$，那么 $x\sim z$。

一个函数 $f:\mathcal{D}_1\rightarrow\mathcal{D}_2$ 是：

一一映射（或单射）　如果对任意两个元素 d_1，$d_2\in\mathcal{D}_1$，$d_1\neq d_2$ 有 $f(d_1)\neq f(d_2)$。

满射　如果对任意元素 $c\in\mathcal{D}_2$ 存在一个元素 $d\in\mathcal{D}_1$ 满足 $f(d)=c$。

双射　如果这个函数既是一一映射又是满射。

2.2 字符串和语言

一个字符串是某个预先定义的集合（称为字母表）中元素的（有限或无限）序列。实践中，通常用符号 Σ 表示字母表，Σ 中的元素叫做字母，某个字母表上的字符串的（有限或无限）集合 L 叫做语言。一个字符串一般可以通过顺序列出其中的全部字母来表示。例如，如果 $\Sigma=\{a, b,$

$c\}$，即英语字母表中的头三个字母，那么 *aabacca* 和 *babacca* 是 Σ 上的字符串。既然一个语言是一个（字符串的）集合，我们可以使用通常的集合表示法来定义它。ϵ（这里 ϵ 不是 Σ 中的字母）表示一个特殊的字符串，即空字符串。空字符串不包含任何字母。下面是几种我们接下来将会使用的可作用于字符串的运算：

联结运算 将两个或更多的字符串以一定的顺序拼接到一起。通常的表示方法是将它们以要求的顺序写到一起，并用操作符“.”分隔。例如，我们可以联结三个字符串 *abba*、*acca* 和 *baa*，写成 *abba. acca. baa*，这样得到的结果是字符串 *abbaaccabaa*。字符串的联结运算还可扩展到字符串集合上，例如 $U.V$ 定义为 $\{u.v \mid u \in U \wedge v \in V\}$。很多情况下联结运算符“.”被省略掉了。

重复运算 如果 σ 是一个字符串，那么 σ^* 是一个字符串的集合（一种语言），其中每个字符串都包含 σ 的零次或多次的重复。例如，语言 $(abac)^*$ 包含字符串 ϵ、*abac*、*abacabac* 等。给定一个语言 L，我们用 L^* 表示通过多次联结 L 中的零个或多个单词所获得的语言。例如，如果 $L=\{ab, bc\}$ 那么 L^* 包含 ϵ，ab，bc，$abab$，$abbc$，$ababbc$，$bcbcab$，等等。要注意的是 L^* 中的字符串不限于取出一个单独的字符串 $\sigma \in L$ 然后重复它。一旦定义了运算符“*”，我们现在就可以将一个字母表 Σ 上的语言 L 表示为 $L \subseteq \Sigma^*$。“*”的一个变体是运算符“+”。它的结果类似于“*”，但是不包含空字符串（即零次重复）。另外一个变体是“ω”运算符，其作用是通过无限多次的重复形成无限长的字符串。

集合运算符 诸如“$\cup$”，“$\cap$”或“$\setminus$”，之类的集合运算符可以用于从给定的语言运算得到一个新的语言。因为语言是字符串的集合，我们同样可以用集合的比较关系“$\subseteq$”、“$=$”和“$\supseteq$”在它们之间进行比较。

如果一个字符串 v 能够被分解为 $u.w$（这里 u 和 w 可以是空字符串），我们把 u 叫做 v 的*前缀*，把 w 叫做 v 的*后缀*。

2.3 图

图是一种常见的表示形式，可以表示一组对象 S，以及连接这些对象的某个（二元）关系 Δ。我们将这样的一个图表示为一个二元组 $\langle S, \Delta \rangle$。$S$ 中的对象通常叫做*节点*或*顶点*。节点通常表示为椭圆或其他形状，每个关联的对 $(s, r) \in \Delta$ 叫做*边*。如果关系 Δ 是对称的，意味着如果对于每一个二元组 $(s, r) \in \Delta$ 同样有 $(r, s) \in \Delta$，那么这个图是*无向的*，边可以用连接表示相应顶点的椭圆之间的直线或弧线表示。如果关系 Δ 是不对称的，那么这个图就是*有向的*，它的每条边 (s, r) 用从表示 s 的椭圆指向表示 r 的椭圆的箭头表示。如果 $e=(s, r)$ 是 Δ 中的一条边，那么 s 是 e 的*源点*，r 是 e 的*终点*。同样我们可以说 e 是 s 的*出边*，是 r 的*入边*。

图 2.1 中表示的是有向图的一个例子。节点的集合是 $\{r_1, r_2, \cdots, r_9\}$。所有的边如下：

$$\{(r_1,r_2),(r_2,r_3),(r_3,r_1),(r_2,r_5),(r_3,r_4),(r_4,r_4),(r_5,r_6),$$
$$(r_6,r_7),(r_7,r_8),(r_8,r_5),(r_5,r_9),(r_9,r_7),(r_4,r_8)\}$$

一条边，例如 (r_4, r_4)，从一个节点（在这个例子里是 r_4）返回到这个节点本身，叫做*自边*或*自循环*。

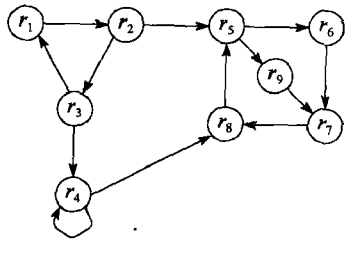


图 2.1 一个有向图

我们可以在图中包含另外一些 S 和 Δ 的成分。一个图的节点、边或两者都可以带有*标签*。给定一个标签的集合 D，一个*边标记函数* $L: \Delta \to D$ 为 Δ 中的每一条边分配 D 中的一个元素。这通常表示为将每条边相应的标签放到靠近这条边的地方。这种情况下，图是一个四元组 (S, Δ, D, L)。标签使我们在遇到某些节

点对之间有不止一条边的情况时可以区分这些边。这些边还可以被重新定义为在 $S\times D\times S$ 之上的三元组，这样每条边就包含源点、标签和终点。在这种情况下对于每条边（s，a，r），边标记函数 L 返回边在中间分量 a 上的投影。

*路径*是 S 中节点的一个（有限或无限）序列，s_0，s_1，s_2，…，s_n，…满足每对相邻的节点 s_i，s_{i+1} 形成一条边（s_i，s_{i+1}）$\in\Delta$。一条路径中如果没有节点出现超过一次，那么就说这条路径是*简单路径*。要注意的是在一个有限图上的一条无限路径中，至少有一个节点必须重复出现无限多次。*环*是一条开始并结束于同一个节点的有限路径。路径的*长度*是指出现在其中的边的数量，包括重复的边（因此，一条路径中的节点数要比这条路径的长度大 1）。因此，对于每个节点而言，都有一条从该节点到它自身的长度为 0 的平凡路径。在图 2.1 所示的图中，有一条长度为 7 的简单路径 r_1，r_2，r_3，r_4，r_8，r_5，r_9，r_7。路径 r_5，r_6，r_7，r_8，r_5 是一个环，长度为 4。在一个图中，两个节点间的*距离*是它们之间最短路径的长度。r_2 和 r_5 间的距离就是 1，r_1 和 r_7 间的距离是 4。

对于图的节点的一个子集 $S'\subseteq S$，如果 S' 中任意一对节点之间都有一条只经过 S' 中节点的路径，那么就说 S' *是强连通的*。*强连通分量*就是这种节点的最大集合，即不可能向这个集合中再加入任何一个节点且仍然保持强连通。图 2.1 中所示的图有三个强连通分量：$\{r_1, r_2, r_3\}$，$\{r_4\}$ 和 $\{r_5, r_6, r_7, r_8, r_9\}$。一个*平凡强连通分量*是指只包含一个节点且没有自循环的强连通分量。

有很多图论算法 [33] 能够用于寻找与有限图相关的重要信息。例如，这些算法能够检验两个节点间是否有一条路径，且返回它们之间的距离。一个对程序自动验证特别有用的算法是 Tarjan 的用于寻找强连通分量的 DFS 算法。另一个重要的算法是计算图的*传递闭包*的 Floyd-Warshall 算法 [145]，图的传递闭包是指节点对（s，r）的集合，使得图中有从 s 到 r 的路径。

*树*是一个无环的有向图，其中有一个没有入边的节点作为树的*根*，并且其他节点都只有一条入边。因此，从根到树中任何一个节点都有一条路径（实际上也是唯一的一条）。节点的*出度*是指其出边的数目。一个常用的绘制树的方法是将根节点放在顶部，然后可以从根节点经过一条边到达的节点水平排列在下一排，然后可以从这些节点中的某个节点经过一条边到达的节点画在再下一排，以此类推，如图 2.2 所示。每一排这样的节点集合称为*一层*。那么，根节点是第 0 层的唯一节点，第 i 层节点有一条长度为 i 的、从根节点出发的路径。如果在树中从节点 r 到节点 r' 有一条路径，那么 r' 叫做 r 的*一个后代*，而 r 是 r' 的*一个祖先*。如果这条路径的长度为 1，那么 r 是 r' 的*直接祖先*，r' 是 r 的*直接后代*。一个没有后代的树节点叫做*叶节点*。

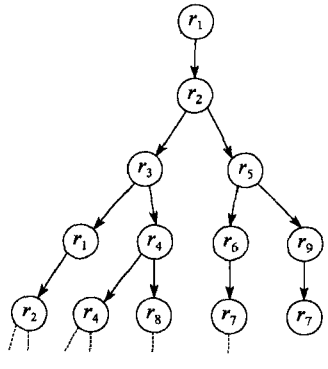


图 2.2 图 2.1 中图的一个展开

有时从一个给定的节点 ι 查看图 G 的*展开*是很有用的。正如下面所描述的，图的一个展开是一棵树，其中的节点用 G 中的节点标记。我们将展开中的每个标记为 s 的节点称为 s 的*一次出现*。在展开中，G 中的一个节点可能会多次出现。这个展开的根节点是 G 中节点 ι 的一次出现。如果在展开的第 i 层有一个节点 s 的一次出现，且 G 中有一条从 s 到 s' 的边，那么在第 $i+1$ 层中必有 s' 的一次出现且必有一条从 s 的出现到 s' 的出现的一条边。假设 s_1 和 s_2 都出现在展开的第 i 层中，且在 G 中两者都有一条到某个节点 s 的边。那么在展开的第 $i+1$ 层中必有 s 的不同出现，其中一个是 s_1 的直接后代，另一个是 s_2 的直接后代。图 2.2 是图 2.1 中从 r_1 开始的无限展开的一部分。

我们形式化地定义 $Tr=\langle\iota, \hat{S}, \hat{\Delta}, L\rangle$ 是图 $G=\langle S, \Delta\rangle$ 的从 S 中某个节点 ι 开始的一个展

开，满足下列条件：

- Tr 是一棵树，根节点为 ι，节点集合为 $\hat{S}$，边集合为 $\hat{\Delta}$，标记函数为 $L:\hat{S}\to S$。
- 如果对 $r\in\hat{S}$ 有 $L(r)=s$，且 G 中 s 有 m 条出边到达节点 s_1，…，s_m，那么 r 在 $\hat{\Delta}$ 中正好有 m 个直接后代，r_1，…，$r_m\in\hat{S}$，使得 $L(r_i)=s_i(1\leqslant i\leqslant m)$。

一个关于树的重要性质是 König 推论（见 [33]）：

每棵无穷树（即有无限多个节点的树）如果其每个节点的出度都是有限的，则必然有一条无限的路径。

2.4 计算复杂度和可计算性

一个算法的复杂度是估计这个算法运行所需要消耗的时间或内存（空间）的一个度量标准。我们也可以讨论一个计算问题的复杂度，这个问题的复杂度体现了解决这个问题的最好算法的复杂度。在一些特定的情况中，复杂度度量标准可以用于评估指定任务所需要的硬件。在其他的一些情况中，用它来表示某个特定算法是不实际的，或者甚至是某个问题根本不可能被有效解决。

因为存在多种多样可用于计算的设备，所以复杂度通常是根据图灵机的抽象模型来度量的[69]。图灵机是计算机程序的一个数学模型，它用线性纸带作为存储设备。纸带被分割为一个接一个的单元，每个单元用给定字母表 Σ 中的某个纸带符号标记，这个字母表包含一个特殊的空白符 $\not b$。纸带的左端固定，并向右无限延伸。读写头能够读取和改变当前纸带位置上的纸带符号，这里的当前位置即是读写头指向的位置（如图 2.3 所示），读写头可以向左或向右移动一个位置，读写头在初始时刻指向最左边的单元。

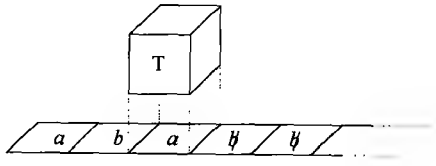


图 2.3 一个图灵机

图灵机的控制机制是一个有限表，这个表为每一对状态和当前输入符号描述了下列内容：(1) 下一个状态；(2) 当前纸带位置的替代符号；(3) 读写头移动的方向，向左或向右。在每一步中，根据当前状态和当前纸带位置上的符号，图灵机能够根据控制表修改当前位置的符号，移动读写头，且改变当前状态。

在执行开始的时候，图灵机的输入处于纸带最左端位置。输入中不包含任何空白符号 $\not b$，且在输入右边跟随着无限连续的空白符号序列。输入规模是指输入所占用的纸带单元的个数。有时，我们会使用一种稍微复杂一点的有多条纸带的图灵机。这样，输入放在一条纸带上，不能重写这条纸带，而其他的纸带被当做存储器使用。图灵机还有很多变体。

图灵机可以作为一种输入单词的接受器。如果计算过程终止于一个被标记为接受的状态，那么图灵机就接受这个单词，如果计算没有终止或者终止于一个被标记为拒绝的状态，那么图灵机拒绝接受这个单词。从另外一个角度看，还可以将图灵机看做是一个转换器，它将输入转换为相应的输出，这个输出在计算结束时保存在纸带上。

图灵机模型是算法的一种非常抽象的表示形式。真正的计算机当然能够进行比移动读写头、改变当前符号与状态更加复杂的操作。然而，图灵机模型仍是一种优秀的、可用于比较不同算法与计算问题的数学度量工具。更进一步讲，虽然真实的计算机的操作要比图灵机的操作复杂得多，但它们执行的计算方法总体上还是相同的。

图灵机的计算模型有几个优点。它相当简单且抽象，剔除了真实计算机需要考虑的很多细节。这也就意味着当一个计算问题或算法从一台真实计算机移植到另一台时，我们不需要再次分析它。这个模型同样允许将一个计算问题视作一种由可接受的输入单词组成的语言。通过这种形式，它与计算机科学中经常使用的其他计算模型相关联，例如有限自动机（见 5.5 节）。事

实上，图灵机可以被认为是一个带有无限存储空间（指无限的纸带）的有限状态自动机（指有限的图灵机控制命令）。

度量一个算法的有效性时既要涉及时间复杂度，又要涉及空间（即存储器）复杂度。时间是指图灵机执行的步数，空间是指用到的纸带符号的个数。（当度量空间时，我们使用上文提到的多纸带的图灵机。这样做的原因是，通过这种方式，空间可以是输入规模的一小部分。）

复杂度度量以输入规模 n 的一个函数的形式给出。我们用“大 O”标记$O(f(n))$ 表示最坏情况下的复杂度，这里 f 是一个函数。例如，$O(n^2)$（表示平方复杂度）或者$O(2^n)$（表示指数复杂度），这里 n 是输入规模。$O(f(n))$ 表示法的解释如下：

> 对于一个算法或者计算问题，如果存在两个常数 c_0 和 c_1，使得对于每一个输入规模 $n>c_0$ 的实例，在图灵机上执行算法（或解决问题）所需要的时间/空间不超过 $c_1\times f(n)$，那么它有$O(f(n))$的复杂度。

这样解释$O(f(n))$ 的原因如下：

- 人们通常不关心一个问题是否有有限多个规模比某个常数 c_0 小、复杂度与 $f(n)$ 不一致的实例的存在。
- 给出的复杂度度量只依赖于某个常数因子 c_1。这是因为我们在计算复杂度时，通常不希望考虑机器的执行速度，或字的大小。如果我们以后买了一台速度更快或者每个内存单元是现在计算机的两倍（比如 64 位，而不是现在的 32 位）的计算机，复杂度仍然保持一样。（实际上，我们可以找到一个足够大的 c_1，就可以消除常量 c_0。但这样会导致 c_1 变得太大。）
- 可能存在一些规模为 n（即使 $n>c_0$）的实例，计算时会比 $c_1\times f(n)$ 更快或者只需更少的存储器开销。但我们只考虑那些规模为 n、造成最坏情况的实例，这是因为我们认为只需要在复杂度度量中考虑最坏的情况。计算机系统的使用者不会因为算法比预期执行得更快或者消耗更少存储器的输入实例而产生不满。

要注意的是如果一个算法的复杂度为$O(f(n))$ 且对每个 $n\geqslant 0$，$f(n)\leqslant g(n)$，那么它的复杂度同样为$O(g(n))$。一个在时间复杂度上最有效的算法通常不一定是在空间复杂度上最有效的算法。

这里我们度量的是算法的*最坏情况复杂度*，即对最坏实例的度量。有时候我们也会对算法的平均复杂度感兴趣，平均复杂度是基于某个概率分布的。实践中，我们很少有实验结果来证实待解决问题的实例以一种特定的方式分布。因此，这种复杂度的意义通常比不上最坏情况复杂度。

本书中，我们将给出所介绍的算法的复杂度。我们通常会避免分析繁复的算法复杂度的过程，而是向读者指出相应的参考文献。这里我们将总结一下复杂度理论中的一些重要事实，这可以帮助我们获得一些有关算法效率的直观感受。

令 $P(n)$ 表示某个多项式，即形如 $a_kn^k+a_{k-1}n^{k-1}+a_{k-2}n^{k-2}+\cdots+a_0$ 的关于 n 的函数，其中 a_0，…，a_k 为整数常数。一个多项式的*次数*是指其中出现的最高的幂值。例如，$3n^7+9n^4+3$ 是一个次数为 7 的多项式。*复杂度类*是一类其复杂性满足某些给定要求的计算问题。下面给出一些主要的复杂度类：

log——复杂度为$O(\log n)$ 的问题。

polylog——复杂度为$O(P(\log n))$ 的问题。

线性——复杂度为$O(n)$ 的问题。

多项式——复杂度为$O(P(n))$ 的问题。

指数 —复杂度为$O(2^{P(n)})$ 的问题。

二重指数——复杂度为$O(2^{2^{P(n)}})$ 的问题。

非初等函数——对于这些问题，不存在一个固定的 k，使得这些问题的复杂度为$O(2^{2^{\cdot^{\cdot^{2^n}}}})$，即塔状的 k 层指数。

使用简单的算术推导就可以证明，对于多项式复杂度，复杂度度量中最重要的因式是最高阶（固定的）指数的因式。根据这个结论，$\mathcal{O}(3n^7+9n^4+3)$ 与 $\mathcal{O}(n^7)$ 是相同的。

为了演示复杂度度量，我们考虑规模 $n=5$，10，50，100 和 500 的问题实例。为简单起见，我们假设在下面所有的例子中常数 c_0 和 c_1 分别是 0 和 1。完成一个 $\mathcal{O}(n)$ 的算法的计算过程所花费的计算时间分别是 5、10、50、100 和 500 个时间单位，这里的时间单位以毫秒为例。对于复杂度度量为 $\mathcal{O}(n^2)$ 的算法，相应的执行时间将分别是 25、100、2 500、10 000 和 250 000 毫秒。对于 $\mathcal{O}(n^3)$，相应的时间分别是 125、1 000、125 000、1 000 000 和 125 000 000。对于指数级复杂度 $\mathcal{O}(2^n)$，这个时间是 32，1 024，然后是一个比 1 125 后跟 26 个 0 还要大的数字，再然后是一个比 3 后跟 150 个 0 还要大的数字，等等。对于二重指数复杂度 $\mathcal{O}(2^{2^n})$，即使是对于第一个例子 $n=5$，都将会花费 4 294 967 296 毫秒，比 7 个星期还要长。

如果买一台更快的计算机，比如一台比我们以前用的计算机快 m 倍的计算机，这将会使得计算过程加快 m 倍，这是一个常数因子。如果我们买了一台这样的机器，就只要花费原来花费时间的 $1/m$ 来解决一个复杂度为 $\mathcal{O}(n)$ 的线性问题，或者可以花费同样的时间来解决一个 m 倍大的问题。然而，如果问题的复杂度为 $\mathcal{O}(n^2)$，我们买了 1 台更快的机器，在同样的时间内我们只能解决一个大 $sqrt(m)$ 倍的问题。在解决指数复杂度的问题时，更快的机器带来的帮助并不是很大。在原来的计算机解决一个规模为 n 的指数级复杂度的实例问题所需的时间内，我们能够解决大 $\log(m)$ 倍的问题[⊖]。因此如果计算能力增加了 100 倍，我们也许可以解决一个规模是原来 8 倍大的问题（这里的 log 是指 $\log_2$）。计算速度增长 1 000 倍将使得我们能够在同样的时间内解决一个规模增大仅 10 倍的问题。就算使用一组计算机也不会带来太大的帮助。如果我们用 100 台计算机而不是 1 台，我们最多以 100 为因数加速执行过程。能这样做的前提条件是我们可以把问题并行化，并且以最优的方式利用这组机器。遗憾的是，情况通常不是这样的。例如，考虑这种情况，我们只能够把算法执行区间中的一半进行并行化，100 台机器将把执行时间减少到原来时间的 $1/2+1/200=101/200$。也就是说，在这种情况下，速度只加速到原来速度的两倍左右。通常来说，只有当我们能够将一个算法的绝大部分都并行化了，我们才能从并行化中受益。

另一种分类方式是*确定性*和*非确定性*的复杂度度量之间的区分。确定性复杂度是指常规的计算模型。在这种模型中，每个点上，程序至多只能选择一种继续处理的方法。需要注意的是，根据当前变量的值，if-then-else 语句和 while 循环都只提供一种选择。

非确定性模型仅仅是一种数学模型。它指的是这样一种执行模型，允许我们在继续执行的路径中作出非确定的选择。对于图灵机这个特定的情况，给定当前状态和当前纸带位置上的符号后，存在不止一个可采用的转换。一个非确定的图灵机接受一个输入的条件是，如果在这个输入所有可能的执行情况中至少有一个是可接受的，那么图灵机就接受这个输入。对于非确定的算法，我们度量的是单一的能终止的计算过程所需要的时间和空间。非确定执行模型在使用一个等价的确定性算法进行模拟时通常会带来时间上的指数级爆炸。能否以一种确定的方式实现众多的非确定性算法且不会带来这种指数级的爆炸，这在目前仍然是一个悬而未决的问题（这也许是计算机科学中最重要的一个问题）。人们怀疑这是不可能的，但还没有人能够形式化地证明这一结论。

这种类型的非确定性只是断言了存在一种在给定时间量内给出正确答案的计算过程。这种非确定性与并发理论中使用的非确定性有所不同，并发中的非确定性将在后面的第 4 章中讨论。

⊖ 这里采用的是原文中的说法，但原文有误。实际上，计算机速度快 100 倍，只能使能够解决的问题规模增加一个常量。假设原来的计算机解决规模为 n 的问题需要 e^n 多的时间，那么只要问题规模比 n 大 ln100（不是 ln100 倍），它花的时间就会增长 100 倍（因为 $100e^n=e^{(n+\ln 100)}$），也就是说快 100 倍的计算机花费同样时间只能够解决问题规模为 $n+\ln 100$ 的问题。因此，对于指数复杂度的问题，即使计算机的速度成倍增加，问题规模的增加也是有限的。这一段中关于指数复杂度的说法都是有问题的。 ——译者注

在并发理论中，非确实性是由进程间通信或者共享变量的使用而引起的。那么，为了使程序能够满足它的规约，我们希望不管作出什么样的选择，程序总是将满足它的规约的。

当然，在实践中我们是不想运行一个非确定的算法的，非确定的算法中有一种计算过程会给出正确的结果，但是其他的计算过程则甚至不能保证能停止。在复杂度理论中使用非确定性仅仅是为了帮助区分一些问题的复杂度。对于某些重要的计算问题，确定性复杂度包含的信息更丰富，使用不确定性复杂度很大程度上说明了我们关于这些确定性复杂度还缺乏更多的认知。我们将遵循计算复杂度的通常意义，且在适当的地方使用确定性或者不确定性类。

复杂度类通常包含下列三个指标：(1) 空间或时间；(2) 度量（如多项式、指数）；(3) 确定或不确定执行模型之间的选择，复杂度类的名称反映了这些选择。考虑下列常见的复杂度类：

NL 非确定性对数空间。要注意的是空间复杂度度量标准不考虑输入规模（因此空间复杂度可以小于线性）。这个类中的算法通常都是很高效的。

P 确定性多项式时间。这个类中的算法被认为是高效的，特别是当多项式的次数较低时。在实践中，这个类中的绝大多数算法都有固定的较小次数的复杂度，例如$\mathcal{O}(n^2)$(平方)，$\mathcal{O}(n^3)$(立方)，因此它们是实用的。虽然如此，一个复杂度为$\mathcal{O}(n^{150})$的算法虽然被认为是多项式复杂度的，但肯定不实用。

NP 非确定性多项式时间。这个类中的问题通常是低效的，因为当把它们转换为确定性算法时会引起指数级的爆炸。然而经验表明，在很多实际案例中存在多种不同的启发式的解决方案能高效解决一些困难的 NP 问题。

PSPACE 多项式空间。(已知的是，确定性和非确定性多项式空间类是相同的 [125]。) 解决这种复杂度的问题需要较大型和快速的计算机。我们仍不知道是否存在多项式算法能够解决所有的 PSPACE 问题。人们猜测这个悬而未决问题的最终回答是否定的。

EXPTIME 确定性指数时间。在实践中只能应用到小规模的实例上。

EXPSPACE 指数空间。这些问题和更高复杂度的问题只有当输入被限制在很小规模的实例上时才可解，比如 $n<10$。

NONELEMENTARY 这里空间和时间都是相同的。有这种复杂度的算法被认为是非常非常困难的。

我们可以观察到类名开头有字母 N 的类对应于一个非确定性度量，而没有 N 时则对应于一个确定性度量。TIME 和 SPACE 有时候会被省略，通常在实践中是指它们其中的某一个，例如 NP 代表了 NPTIME。

已知

$$\mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME} \subseteq \mathrm{EXSPACE}$$

其中某些类之间存在严格的包含关系，例如 P⊂EXPTIME，但我们不知道 P⊂PSPACE 是否成立。

一个计算问题的复杂度度量可以通过上界或下界的形式给出。上界是指解决这个问题的某些具体算法能达到的一个已知最好的复杂度。这个问题仍然可能存在一个更好的算法（尽管还是未知的）。一个计算问题的下界说明了解决这个问题的难度至少是这个复杂度。但是，并不需要存在一个满足下界复杂度的具体算法。为了说明下界，我们需要定义 Ω 表示法 $\Omega(f(n))$ 如下：

> 一个算法或者计算问题的复杂度是 $\Omega(f(n))$ 仅当存在两个常数 c_0 和 c_1，使得对于每个输入规模 $n>c_0$ 的实例，在图灵机上运行这个算法（解决这个问题）所需要的时间/空间至少是 $c_1 \times f(n)$。

如果一个问题的下界和上界相同，那么我们说它是一个紧致的复杂度界；否则就存在一个需要进一步研究的复杂度差距。对于计算机科学中的许多基本问题，紧致的复杂度界仍然是未知的。

有些有趣的问题类具有相同的复杂度差距。这样的一个类中的问题通过*归约*相互联系。归约是一种“高效的”直接转换，例如，转换过程在时间上是多项式复杂的（不同的复杂度类可能对效率有不同要求）。一个问题对于一个复杂度类是*完备*的，当且仅当它属于这个类且类中每个其他的问题都能有效地归约到这个问题。直觉上，一个完备的问题至少和该类中的其他问题一样难。这样的复杂度类的例子包括 PSPACE 完备和 NP 完备。PSPACE 完备中的问题能够用多项式空间解决，而 NP 完备中的问题能够在非确定性多项式时间内解决。没有人知道这些类中任何一个问题的紧致复杂度下界是什么。此外，如果一旦有人发现了某个类中至少一个问题的确定性多项式时间算法，那么这一发现将会自动地为该类中的所有其他问题提供多项式时间算法。

虽然 NONELEMENTARY 复杂度看上去已经足够坏了，但它还不是最坏的情况。*可计算性理论* [69] 证明了有些问题不可能用任何算法解决。我们说这些问题是*不可判定的*。这样的问题通常很一般，例如，确定一个程序能否在每个合法的输入上都停止。尽管这最初看起来好像是一个很理论化的问题，但这个问题对软件可靠性方法的打击很大。保证一个给定的程序能够终止当然是我们想验证的属性之一。

诸如检验一个问题在给定的输入上能否终止，这样的问题是*半可判定的*（也可以叫做*递归可枚举的*）。这意味着存在一个算法，当肯定的答案存在时，该算法终止并给出正确答案，但是当回答为否定时，这个算法不一定能够停止。对于检验一个算法在某个输入上是否能够终止这一具体问题，我们可以简单地在给定的输入上模拟程序运行，然后观察模拟过程是否停止。半可判定的判定问题并没有相关的复杂度度量。如果存在这样的度量，我们就可以创建出一个判定过程，而这和问题的半可判定性矛盾。假设要构造这样的判定过程，我们可以执行半判定过程，直到要么它停止，要么由复杂度度量规定的时间被用尽。在出现后一种情况时，我们可以知道答案是否定的。

正如后面的章节将要展示的，复杂度和不可判定性因素对软件可靠性方法的应用能力有重要的影响。不可判定性结果对演绎验证有严重的后果。它表明我们想要为实际系统验证的绝大部分属性，诸如停机、服务的授予等，都不能以算法的方式完全验证。正如我们将看到的，这并不能阻止训练有素的工程师、数学家或逻辑学家给出形式化的证明。但是，验证过程就不再具有时间限制。不管怎样，在实践中，我们通常只尝试证明那些按照充分理解的原理运行的系统的正确性。在这种情况下，对于这些方法的有经验的使用者，应该能够利用开发人员的直觉对系统进行验证。

某些计算问题的高复杂度，甚至是不可判定性，都不应该阻止研究人员去尝试发现启发式的解决方案，即能够高效解决某些重要的实际案例的解决方法。

2.5 扩展阅读

有关集合论的教材包括：

P. R. Halmos, *Naive Set Theory*, Springer-Verlag, 1987.
K. Devlin, *The Joy of Sets: Fundamentals of Contemporary Set Theory*, 2nd edition, Springer-Verlag, 1993.

有关图论的书包括：

R. Diestel, *Graph Theory*, Springer-Verlag, 2nd edition, 2000.
A. Gibbons, *Algorithmic Graph Theory*, Cambridge University Press, 1985.

有关可计算性和复杂度理论的书包括：

J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, 1979.
H. R. Lewis, Ch. Papadimitiou, *Elements of the Theory of Computation*, Prentice-Hall, 2nd edition, 1997.
Ch. Papadimitiou, *Computational Complexity*, Addison-Wesley, 1994.
M. Sipser, *Introduction to the Theory of Computation*, PWS, 1996.

第3章
Software Reliability Methods

逻辑和定理证明

"正好相反，"特威度迪接着说，"说过去是这样，也有可能；假设是这样，也能自圆其说；但是事实不是这样，就不是这样。这就是逻辑。"

刘易斯·卡洛尔《爱丽丝镜中奇遇记》

数学逻辑提供了软件验证方法的基础。就像是程序设计语言，逻辑结合了语法和语义：语法规定如何来写合法的公式，语义则为每个公式给出明确的含义。数学逻辑将证明的概念形式化。本章中，我们将综述一阶逻辑和命题逻辑，之后探讨机械化的定理证明的精髓。定理证明工具通常不能完全自动化地得到证明，而是在证明的过程中保证严密性并提供证明指引，为用户提供帮助。在后面的章节中，我们会展示各种逻辑和证明系统，它们可以通过手动、计算机辅助或完全自动化等方式来证明程序的属性。

3.1 一阶逻辑

一阶逻辑（first-order logic）这个形式化方法在数学中常用到，例如在微积分和几何中；在计算机科学领域里也常用到，例如在数据库和人工智能中。我们在第7章将会看到，一阶逻辑在程序验证中大量使用。谓词，基本上是一阶逻辑公式的一种受限制的形式，它们通常在 if-then-else 语句或者 while-loop 中出现。一阶逻辑使用变元（variable）、关系（例如"≤"）和函数（例如"×"或"+"），其中变元的值域是某个指定的论域（domain），例如整数或实数。一阶逻辑可用于推导该域中的所有对象，或者断言存在一个对象满足一个属性。

一个逻辑被称为是一阶的原因是其公式中的所有变元的值域都是某个预先指定的论域（例如整数）。在二阶逻辑（second-order logic）中，则既可使用简单变元也可使用集合变元，而后者的取值可以是对象的集合。这样，如果 x 是一个简单变元而 Y 是一个集合变元，就可以在二阶逻辑中表达一个公式 $x \in Y$，其含义是变元 x 所代表的值包含在变元 Y 所代表的集合中。而三阶逻辑则包含更为复杂的变元，代表集合的集合，以此类推。

一阶逻辑构成了数据库理论的理论基础 [2]。程序设计语言 Prolog [32] 及衍生的 Datalog 的语义和语法就来自一阶逻辑，其执行机制则基于逻辑推理。使用基于逻辑的程序设计语言能自动地提供更高层次的可靠性，因为程序可以被看做是其自身的规约。然而，这并不能保证这样的程序一定能够终止。这个属性需要另作证明。

3.2 项

一阶项（term）是根据一阶逻辑的精确语法来表示的表达式。我们将首先定义项的语法，然后给出其语义解释。一个签名（signature）$\mathcal{G}=(V, F, R)$ 包括三个不相交的集合：变元符号集合 V、函数符号集合 F 和关系符号集合 R。这些仅是语法对象，而并无先验的特殊含义。因为我们只能使用有限数量的字符，所以很难为每个函数或关系使用不同的字符。因此，我们经常使用程序设计语言的语法来表示符号：由一个字母开头，之后是字母、数字和下划线"_"的任意组合。从现在开始，我们假定可以从上下文推断出一个符号表示的是变元、函数或是关系。

我们接下来为属于某个签名的每个函数或关系符号指定元数（arity），也就是其参数的个数。

例如，加法函数 *add* 的元数是 2，*sine* 函数的元数是 1，而关系 *ge*（表示“大于或等于”）的元数是 2。常量符号，例如 *zero* 或 *one*，则被看做是 0 元的函数符号。

项就是用函数符号和变元构造的表达式。例如，令 *v*1 为一个变元，*zero* 和 *one* 是常数，*add* 是二元的函数符号。那么，*add*（*one*，*one*）和 *add*（*add*（*one*，*one*），*v*1）就是项。我们用在定义程序设计语言结构时经常用到的 BNF 记法来定义项的语法：

$$term ::= var \mid const \mid func(term, term, \cdots, term)$$

其中 *var* 是变元符号，*func* 是函数符号，而 *const* 是常量符号。在 BNF 中，“::=”表示“定义为”，而竖直线用来分隔定义中不同的选择。这样，一个项可包含：（1）变元符号；（2）常量符号；（3）函数符号后面跟着左括号，接着是由逗号分隔的项的列表，最后是右括号。在（3）中，每个项又是由同一个 BNF 公式来递归定义的。在这个 BNF 定义之外，我们还有如下约束：在一个 n 元函数符号后的括号中必须有 n 个项，并用逗号分隔。

相应地，*v*1 是一个变元，因此是一个项；*one* 是一个常量，因此是一个项；*add*(*one*，*one*) 也是一个项，它将二元函数符号应用到两个项上（我们已经说明它们都是项）；*add*(*add*(*one*，*one*)，*v*1) 还是一个项，它再次将函数符号 *add* 应用在前面的两个项上。在下面的讨论中，我们放宽这些记法，允许使用常见的数学和程序设计语言函数以及常量符号，例如“+”或 175。此外，我们还允许使用中缀表达式，也就是说，将关系符号放在两个操作数之间，例如 *v*1+1。

到目前为止，我们还没有给这些定义的对象赋予其特定的含义（语义）。让项 *add*(*add*(*one*，*one*)，*v*1) 表示变元 *v*1 加 2 的值看起来是很直观的。然而，仍然需要形式化地定义写下的符号和期望的含义之间的联系。为了解释项，我们首先假定一个给定的论域，也就是一个值的集合 $\mathcal{D}$。例如，这个论域可以为：

- 整数。
- 自然数，也就是正整数和 0。
- 有理数，也就是能够由两个整数相除得到的数（除数不能为 0）。
- 定义在某个字母表上的串。

一阶结构是在一个给定签名的基础上定义的。结构 $\mathcal{S}=(\mathcal{G}, \mathcal{D}, \mathcal{F}, \mathcal{R}, f)$ 包括签名 $\mathcal{G}=(V, F, R)$、论域（也就是集合）$\mathcal{D}$、函数集合 $\mathcal{F}$（包括常量）、关系集合 $\mathcal{R}$，以及映射 $f: F \cup R \rightarrow \mathcal{F} \cup \mathcal{R}$。它把签名中的函数和关系符号分别映射到论域 $\mathcal{D}$ 上的实际函数和关系。举例来说，函数符号 *sub* 可由 f 映射到整数域上的减法。关系符号 *ge* 可由 f 映射成一个关系，该关系包含所有第一个整数大于或等于第二个整数的整数对。映射 f 必须保持函数和关系的元数不变。也就是说，f 将元数为 n 的函数符号映射到带有 n 个参数的函数上，将元数为 n 的关系符号映射到带有 n 个参数的关系上。

逻辑的语法规定了如何在一张纸上或计算机内存中表示一个函数或关系符号，以及更为复杂的对象，比如项和公式。语义则把这些记号连接到（或者说映射到）数学对象。相应地，一阶逻辑的签名规定了可以用于指代函数或关系的语法对象（符号）；逻辑结构中的分量 $\mathcal{D}$、$\mathcal{F}$ 和 $\mathcal{R}$ 则用于提供对应的数学对象。语法解释函数 f 则用来把签名中的语法对象和对应的数学对象联系起来。

举例来说，f 可将函数符号 *add* 映射到整数上的加法，我们通常用“+”表示。注意“+”本身也只是一个语法对象，一个可以打印出来的符号。我们通常为“+”附加了加法的含义。这个符号事实上被重载了，它表示诸如整数、实数、有理数以及复数上的加法。我们通常假设通过上下文可以确定恰当的论域。

3.2.1 赋值和解释

赋值（assignment）a 将变元集合 V 中的变元映射到论域 $\mathcal{D}$ 中的值，我们用 $a: V \rightarrow \mathcal{D}$ 来表示。

例如，如果$\mathcal{D}$是整数集，变元集合V是$\{v1, v2, v3\}$，我们可以得到赋值$a=\{v1\mapsto 3, v2\mapsto 0, v3\mapsto -5\}$。本书中，我们讲的逻辑（或函数）赋值和程序设计语言中的赋值不同：前者将集合中的每个元素映射为某个论域上的值，后者的作用是在程序运行时改变某个指定变元的取值。

我们用$terms(\mathcal{G})$表示签名$\mathcal{G}$上的所有项。现在我们可以定义*语义解释*（semantic interpretation）$T_a: terms(\mathcal{G})\to\mathcal{D}$，它将每个项映射为论域中的值。需要注意的是，该解释仍然依赖于赋值a，其定义是递归的：

$$T_a(v) = a(v), \text{对于 } v \in V$$

$$T_a(func(e_1, e_2, \cdots, e_n)) = f(func)(T_a(e_1), T_a(e_2), \cdots, T_a(e_n))$$

第一行定义了变元的解释是其在赋值a下的值。第二行是项的递归定义。一个形如$e=func(e_1, \cdots, e_n)$的项e的解释可以按照如下方式得到：我们首先得到$f(func)$，也就是在一阶结构$\mathcal{S}$上$func$关联的实际函数。因为项$e_1, \cdots, e_n$比e短，假设我们已经知道如何递归地应用T_a解释它们，并得到$T_a(e_1), \cdots, T_a(e_n)$。然后我们在这$n$个值上应用$f(func)$，得到$e$的解释。

考虑如下的例子，整数域和赋值$a=\{v1\mapsto 2, v2\mapsto 3, v3\mapsto 4\}$。令$f$将$add$映射到整数上的加法。我们有：

$$T_a(v1) = a(v1) = 2$$

$$T_a(v2) = a(v2) = 3$$

$$T_a(v3) = a(v3) = 4$$

$$T_a(add(v1, v2)) = f(add)(T_a(v1), T_a(v2)) = 2 + 3 = 5$$

$$T_a(add(add(v1, v2), v3)) = f(add)(T_a(add(v1, v2)), T_a(v3)) = 5 + 4 = 9$$

将语法和语义费尽心机区分开的做法，一开始看起来很复杂而且似乎没必要。但是，这个区分是非常有用的：通过指明记法（语法）和含义（语义）之间的关系，一阶逻辑允许我们证明论域上的属性。这样的证明是基于语法的，因为它通过对（写在纸上或是表示在计算机内存中的）公式进行字符串操作来完成证明。然而，使用语义解释，这些串操作的结果可立即被投射回所讨论的数学对象。

3.2.2 多个论域上的结构

对软件系统进行推理可能需要多个论域。例如，我们可能需要同时在整数和字符串上进行推理。在这样的情况下，在恰当的参数类型上应用函数和关系时必须小心。在这个扩展下，每个变元可能被赋予一个特定论域中的值。

函数和关系符号可能和多个论域相关。函数和关系的每个参数，以及函数的结果，都必须确定在某个论域上。举例来说，我们可以定义字符串论域上的函数*length*，其值是一个字符串的长度，它是一个自然数。对于多论域一阶逻辑的更多细节，可参见其他文献，如[38]。

3.3 一阶公式

我们仍然首先介绍语法。在项上应用关系符号即构造得到简单公式，其语法为：

$$simp\_form ::= rel(term, term, \cdots, term) \mid term \equiv term$$

其中rel是关系符号。举例来说，$ge(add(one, one), zero)$是一个简单公式，因为二元关系ge应用在两个项$add(one, one)$和$zero$上。我们总是将特别关系“$\equiv$”包括进来，它表示“等于”。

一阶公式包括了如上所述的简单公式。另外，它们还可通过递归地应用布尔运算符“$\wedge$”（和）、“$\vee$”（或）、“$\neg$”（非）以及“$\rightarrow$”（蕴含），或在一个公式前面加上一个量词（quantifier）

“∀”（对于所有）或“∃”（存在）及一个变元名得到。如下所示：

$$form ::= simp\_form \mid (form \wedge form) \mid (form \vee form) \mid (form \rightarrow form) \mid (\neg form) \mid \forall var(form) \mid \exists var(form) \mid true \mid false$$

例如，

$$(ge(one, zero) \wedge ge(add(one, one), v1)) \tag{3.1}$$

是一个公式。同样，$\forall v1(\exists v2(ge(v2, v1)))$ 也是公式。如果公式中的某一部分本身也是一个公式，那么这一部分称为原公式的子公式。这样 $ge(one, zero)$、$ge(add(one, one), v1)$ 以及式（3.1）自身都是式（3.1）的子公式。

与很多程序设计语言类似，布尔运算符之间的优先级可以帮助我们省略一些括号。例如，否定符号“¬”通常比合取符号“∧”的优先级高，而合取符号“∧”又比析取符号“∨”的优先级高。布尔运算符“∧”和“∨”是左结合的，也就是说 $v_1 \vee v_2 \vee v_3$ 和 $((v_1 \vee v_2) \vee v_3)$ 相同。（更仔细地观察布尔运算符的语义就会发现，多个合取或析取的结果与括号的位置无关。）我们总是可以忽略最外层的括号。

也允许使用常见的数学或程序设计语言中的关系符号，例如“≥”，并使用中缀记法。相应地，式（3.1）可表示为

$$1 \geqslant 0 \wedge 1+1 \geqslant v1 \tag{3.2}$$

在形如 $\mu_1 \wedge \mu_2 \wedge \cdots \wedge \mu_n$ 的公式或子公式中，我们说每个 μ_i 都是一个合取项（conjunct）。类似地，在形如 $\nu_1 \vee \nu_2 \vee \cdots \vee \nu_m$ 的公式或者子公式中，我们说每个 ν_i 都是一个析取项（disjunct）。在后面的章节中，我们也使用这样的表示方法 $\bigwedge_{i=1,n} \mu_i$ 和 $\bigvee_{i=1,m} \nu_i$。

现在我们将描述一阶公式的语义解释。首先，关系符号“≡”是标准的“等于”关系。这样，$v1 \equiv v2$ 成立当且仅当 $v1$ 和 $v2$ 被赋予相同的值，即当 $a(v1) = a(v2)$ 时成立。此后，我们将要注意，在逻辑内使用“≡”符号表示相等，而在逻辑外使用“=”表示对象之间的相等。这个区别在逻辑形式系统中很常见，我们必须小心避免混淆目标逻辑内部的断言和关于这个逻辑的断言。然而，很多逻辑书籍不区分符号“=”和“≡”。（在程序上下文中，我们使用通常的程序设计语言的语法，用“=”表示相等。）关于逻辑的断言通过自然语言（例如英语）表示，我们通常称之为元语言（meta language）。

回顾一下，f 是从关系符号到关系的保持元数的映射，例如，leq（小于或等于）被映射为整数对之间的标准关系，通常用“≤”表示。令 $forms(\mathcal{G})$ 表示签名 $\mathcal{G}$ 上的一阶公式集合。解释映射 $M_a : forms(\mathcal{G}) \rightarrow \{\text{TRUE, FALSE}\}$ 将布尔值（或者说真值），也就是 TRUE 或 FALSE，赋给每个公式。解释仍然依赖于赋值 a。我们首先给出不包含量词“∀”和“∃”的公式的解释。

$$\begin{aligned}
M_a(pred(e_1, \cdots, e_n)) &= f(pred)(T_a(e_1), \cdots, T_a(e_n)) \\
M_a(e_1 \equiv e_2) &= (T_a(e_1) = T_a(e_2)) \\
M_a(f1 \wedge f2) &= \text{TRUE iff}(M_a(f1) = \text{TRUE and } M_a(f2) = \text{TRUE}) \\
M_a(f1 \vee f2) &= \text{TRUE iff}(M_a(f1) = \text{TRUE or } M_a(f2) = \text{TRUE}) \\
M_a(f1 \rightarrow f2) &= \text{TRUE iff}(M_a(f1) = \text{FALSE or } M_a(f2) = \text{TRUE}) \\
M_a(\neg f1) &= \text{TRUE iff}(M_a(f1) = \text{FALSE}) \\
M_a(true) &= \text{TRUE} \\
M_a(false) &= \text{FALSE}
\end{aligned}$$

公式语义解释的第一行应按如下方式理解：首先，我们用项的解释函数 T_a 解释项 e_1，…，e_n，得到值 $T_a(e_1)$，…，$T_a(e_n)$。现在，$f(rel)$ 是对应语法符号 rel 的关系，这个关系应用到上面的值，可得到 TRUE 或 FALSE。

常量关系符号 *true* 和 *false* 分别被解释为 TRUE 和 FALSE（依据上面语义定义的最后两行）。我们本来可以直接使用布尔值 TRUE 和 FALSE（正如有些书籍中就采用这样的非形式化的方法）。然而，我们小心地区分了语法常量和语义值。注意我们可以用 $\varphi \vee \neg \varphi$ 来替代 *true*，从而在语法中消去 *true*，这里 φ 可以是任意公式；类似地，我们也可以用 $\varphi \wedge \neg \varphi$ 来替代 *false*。（注意记法 iff，读作“当且仅当”，其意思就是“其确切条件是”。）

注意简单公式的解释中使用了项解释函数 T_a。同时，上面公式的语义解释中的第二行，即关于 $e_1 \equiv e_2$ 的部分，包括了三个表示等价的符号：第一个是“$\equiv$”符号，是我们在逻辑中表示等价的一阶符号；第二个，用“=”表示，其含义是左边的一阶项的解释通过右边的等式定义；第三个，也用“=”表示，要求两个计算值之间相等。

在赋值 $a=\{v1 \mapsto 2,\ v2 \mapsto 3,\ v3 \mapsto 4\}$ 下解释公式

$$\varphi = ge(add(add(v1,v2),v3),v2) \wedge ge(v3,zero)$$

可按照如下方式进行：

$$\begin{aligned}
T_a(add(add(v1,v2),v3)) &= 9(\text{如 3.2 节所示}) \\
T_a(v2) &= 3 \\
T_a(v3) &= 4 \\
T_a(zero) &= 0 \\
M_a(ge(add(add(v1,v2),v3),v2)) &= f(ge)(T_a(add(add(v1,v2),v3)),T_a(v2)) \\
&= 9 > 3 = \text{TRUE} \\
M_a(ge(v3,zero)) &= f(ge)(T_a(v3),T_a(zero)) \\
&= 4 \geqslant 0 = \text{TRUE} \\
M_a(\varphi) &= \text{TRUE}
\end{aligned}$$

为了给出带有存在量词（“$\exists$”）和全称量词（“$\forall$”）的公式的语义定义，我们首先定义变体（variant）这一概念。令 a 是赋值，v 是变元，而 d 是给定论域$\mathcal{D}$中的一个值。变体 $a[d/v]$ 表示的赋值和 a 相同只是将值 d 赋给了变元 v。形式化定义如下：

$$a[d/v](u) = \begin{cases} a(u) & \text{如果 } u \neq v \\ d & \text{如果 } u = v \end{cases}$$

带量词的公式的语义如下定义：

$$M_a(\forall v(\varphi)) = \text{TRUE iff 对}\mathcal{D}\text{中的每个 } d, M_{a[d/v]}(\varphi) = \text{TRUE}$$

$$M_a(\exists v(\varphi)) = \text{TRUE iff 在}\mathcal{D}\text{中存在 } d \text{ 使得 } M_{a[d/v]}(\varphi) = \text{TRUE}$$

第一行定义指出：$\forall v(\varphi)$ 在某赋值 a 下是 TRUE 的充要条件是对于任何仅在变元 v 的取值上和 a 不同的变体，φ 都是 TRUE。我们把这个公式读作：“对于 v 的所有值，φ 成立”；类似地，$\exists v(\varphi)$ 读作：“存在 v 的一个值使得 φ 成立”。

当结构$\mathcal{S}$可以由上下文推出时，$M_a(\varphi)$=TRUE 通常被记为 $a \models^{\mathcal{S}} \varphi$，读作“在结构$\mathcal{S}$下 a 满足 φ”。如果对于每个赋值 a 都有 $a \models^{\mathcal{S}} \varphi$，那么$\mathcal{S}$是 φ 的模型，并记为$\models^{\mathcal{S}} \varphi$。如果对于每个$\mathcal{S}$都有$\models^{\mathcal{S}} \varphi$，那么 φ 称为重言式（tautology，又称永真式），并记为$\models \varphi$。如果对于任何赋值 a 和任何结构$\mathcal{S}$，$a \models^{\mathcal{S}} \varphi$ 都不成立（我们用 $a \not\models^{\mathcal{S}} \varphi$ 表示），那么 φ 是矛盾式（contradiction）。注意，φ 是重言式当且仅当$\neg \varphi$ 是矛盾式。

上面对“$\models$”符号的三处使用之间的细微差别很重要。我们通过如下的例子来解释：

- 考虑 $a \models^{\mathcal{S}} x \equiv y \times 2$，其中$\mathcal{S}$是包含了整数域的结构，而二元函数“×”解释为乘法。对于例如 x 为 6、y 为 3 的赋值 a，$x \equiv y \times 2$ 成立。然而，$x \equiv y \times 2$ 并不总是成立，比如当 a 赋值 x 为 2 而 y 为 3 时。

- 考虑$\models^{\mathcal{S}} x\times 2\equiv x+x$，其中$\mathcal{S}$是包含了整数域的结构，而二元函数“×”和“+”分别解释为整数相乘和相加。无论实际上什么值被赋给了变元x，一阶属性$x\times 2\equiv x+x$在这个结构中都成立。因而，$\mathcal{S}$是这个公式的模型。
  从另一方面来说，$\models x\times 2\equiv x+x$则不成立。举例来说，给定整数域上的模型$\mathcal{S}'$，其中“×”表示在其参数的二进制形式（例如常数2表示为0010）上的位与运算符，而“+”表示位或运算符。
- 最后，考虑$\models (x\equiv y\wedge y\equiv z)\rightarrow x\equiv z$。后一个公式表示了标准等价关系的传递性。它不依赖于结构或赋值。因此，它是一个重言式。

在一阶公式中，变元v在形如$\forall v(\varphi)$或$\exists v(\varphi)$的子公式中的任何出现，叫做受限（quantified）的出现，任何其他的出现称作自由出现。根据上面的讨论，我们有，如果$\models^{\mathcal{S}}\varphi$或$\models\varphi$时，$\varphi$的真值不依赖于$\varphi$中出现的自由变元的值。因此，这些情况下，$\models^{\mathcal{S}}\varphi$或$\models\varphi$分别和$\models^{\mathcal{S}}\varphi'$或$\models\varphi'$等同，其中$\varphi'$是在$\varphi$的每个自由变元上应用全称量词得到的——举例来说，$\exists x(y<x\wedge x<z)$可以写成$\forall y\forall z\exists x(y<x\wedge x<z)$。事实上，如果$\varphi$不包含自由变元，那么$\mathcal{S}$下$\varphi$的真值不依赖于任何赋值$a$。这样，如果$a\models^{\mathcal{S}}\varphi$对于某个赋值$a$成立，那么$\models^{\mathcal{S}}\varphi$成立。反过来说，根据定义，如果$\models^{\mathcal{S}}\varphi$成立那么对于每个赋值$a$，$a\models^{\mathcal{S}}\varphi$都成立。

有趣的是，给定一个结构$\mathcal{S}$，对于每个一阶公式φ来说，要么$a\models^{\mathcal{S}}\varphi$成立，要么$a\models^{\mathcal{S}}\neg\varphi$成立。当$\varphi$中没有自由变元时，我们还可以得出结论：要么$\models^{\mathcal{S}}\varphi$成立，要么$\models^{\mathcal{S}}\neg\varphi$成立。另一方面，如果$\varphi$既不是重言式又不是矛盾式，那么可能$\models\varphi$和$\models\neg\varphi$都不成立。

为了说明逻辑的语法和语义之间的区别，考虑如下的公式：

$$\varphi = \forall v1\forall v2(v1<v2\rightarrow\exists v3(v1<v3\wedge v3<v2))$$

如果没有进一步了解公式在什么结构下进行解释，我们就不能说这个公式成立还是不成立。尽管我们可以猜测“<”表示“小于”关系，但不能保证每个结构都是这种情况。此外，即使“<”真的是“小于”，我们也还需要注意这个关系可以在不同的论域上解释。假如论域是实数，那么这个公式表示的就是，对于每对赋给$v1$和$v2$的值，若$v1$比$v2$小，那么必然有一个实数值$v3$在$v1$和$v2$之间。这个公式对于实数恰好是成立的。现在，假设论域是整数，那么这个公式就不成立。因为两个相邻整数（例如2和3）之间并没有其他整数。

对于公式φ、项e和变元v，定义$\varphi[e/v]$表示通过在φ中用项e替换v的每次自由出现而得到的公式。（注意受限出现不被替换。）进一步地，我们不允许将包含v的某个自由出现的项替换为包含一个在替换处受限的变元的项。下面的例子解释为什么需要这个限制。考虑公式$\forall x(x>y\rightarrow x>z)$。当该公式在整数上解释，并且“>”解释为“大于”时，这个公式表示y大于或等于z。现在，如果我们允许用$x-1$替换y（x在替换处受限——译者注），我们就得到$\forall x(x>x-1\rightarrow x>z)$。因为$x>x-1$对于所有的整数都成立，所以这个公式断言任意数都比$z$大。这是一个错误的命题，因为整数中没有最小数。

注意，被替换的变元仍可以包含在替换它们的项中。因此，在把该变元替换为相应的项之后不能再重复进行替换。例如，若$\varphi=v1\geqslant v2$，那么$\varphi[v1+1/v1]$就是$v1+1\geqslant v2$。如果没有上述考虑，这个替换过程就不会停止。

对赋值和公式的替换使用相似的表示方式，这并不是偶然的。通过对公式的大小进行归纳（参见其他文献，如[38]），我们可以证明对于每个一阶公式φ、项e和变元v，只要可以进行替换$\varphi[e/v]$，我们有，

$$a[T_a(e)/v]\models^{\mathcal{S}}\varphi \text{ 当且仅当 } a\models^{\mathcal{S}}\varphi[e/v]$$

这一联系说明了语法替换和语义替换之间的关系。在语义方面（左侧），我们令v的值为表达式e（在原赋值a下）的值$T_a(e)$，得到赋值a的变体。在语法方面（右侧），我们用表达式e替换变元v的每次出现。

我们有时写成 $\varphi(v_1, v_2, \cdots, v_n)$，以强调变元 $v_1, v_2, \cdots, v_n$ 是公式 φ 中的自由变元。相应地，我们可以用 $\varphi(e_1, e_2, \cdots, e_n)$ 表示将 φ 的自由变元 v_i 替换为表达式 e_i 而得到的公式。这样，我们可写成 $\varphi(y, z) = \exists x(y<x \wedge x<z)$。那么 $\varphi(3, t+4)$ 就是 $\exists x(3<x \wedge x<t+4)$。

当 $\models \varphi \rightarrow \psi$ 或 $\models^S \varphi \rightarrow \psi$ 时，我们说 φ 比 ψ *强* (stronger)，或 ψ 比 φ *弱* (weaker)。用结构的概念来说，$\models \varphi \rightarrow \psi$ 表示在所有使 φ 成立的结构和赋值（可能还有一些其他的结构）下，ψ 成立。当 φ 比 ψ 强，同时 ψ 也比 φ 强时，我们说 φ 和 ψ 是*逻辑等价的* (logically equivalent)。

在看比较老的逻辑书时，有一点容易引起误解。这些书中，蕴含符号“→”有时写作“⊃”。而这个符号现在通常被用于集合包含。在 $\varphi \supset \psi$ 中使用这个记法，恰恰和以下事实相反：使得 φ 成立的结构的集合包含在使得 ψ 成立的结构的集合中。也就是说，如果 $\models \varphi \rightarrow \psi$，那么 $\{\mathcal{S} \mid \models^S \varphi\} \subseteq \{\mathcal{S} \mid \models^S \psi\}$。

表达能力

形式化方法非常重要的一个属性就是其*表达能力* (expressiveness)。表达能力指的是这个形式化方法表述说明的能力。考虑给定的签名 $\mathcal{G}$。令 $\mathcal{H}$ 表示 $\mathcal{G}$ 上具有我们想要描述的特别属性的结构的集合，而 $\mathcal{G}$ 上其他的结构不满足这个属性。我们说一个逻辑能够*表达*这个属性，如果该逻辑允许写出一个在 $\mathcal{G}$ 上的公式 φ，使得 φ 恰恰对 $\mathcal{H}$ 中的结构成立。

举例来说，考虑一个包含二元关系 R 的签名，该关系代表一个图。我们可以用一阶逻辑描述下面的属性：

- 图是无向的（即关系 R 是对称的）。

$$\forall x \forall y(x\ R\ y \rightarrow y\ R\ x)$$

- 没有孤立节点，即所有的节点都和其他节点相连。

$$\forall x \exists y(x\ R\ y \vee y\ R\ x)$$

另一方面，也有一些我们感兴趣的属性是不能用一阶逻辑来表示的。

- 图是有限的。
- 图中有圈 (cycle)。
- 图是连通的（即只包含一个强连通分量）。

3.4 命题逻辑

命题逻辑 (propositional logic) 是比一阶逻辑简单的形式化方法，它不包含量词，也不允许函数和关系符号（包括等于符号“=”），它有一组*命题变元* AP。每个这样的变元的值域都是布尔值 {TRUE, FALSE}。该逻辑的语法定义是：

$$form ::= prop \mid (form \wedge form) \mid (form \vee form) \mid$$
$$(form \rightarrow form) \mid \neg form \mid true \mid false$$

其中 $prop$ 是变元集合 AP 的命题变元。这里的赋值 a 重新定义为 $a : AP \rightarrow \{\text{TRUE, FALSE}\}$，即 AP 中的每个命题变元都被映射到一个布尔值。布尔运算符的语法和一阶逻辑中的定义相同。命题逻辑中没有涉及签名或结构。因此，当 $M_a(\varphi) = \text{TRUE}$ 时，我们可以写成 $a \models \varphi$，并说 a 满足 φ。如果任何赋值都满足命题公式，则该公式是重言式；如果没有赋值能够满足某命题公式，则该公式为矛盾式。很容易就可以证明，公式的真值仅依赖于对公式中出现的变元的赋值。

举例来说，令 P、Q 为命题变元。公式 $P \wedge \neg P$ 就是一个简单的矛盾式，任何赋值都会给出 FALSE 的解释。类似地，$P \vee \neg P$ 是重言式，它在每个赋值之下都解释为 TRUE。公式 $\varphi = \neg P \wedge (Q \vee P)$ 既不是重言式，也不是矛盾式。在赋值 $a = \{P \mapsto \text{FALSE}, Q \mapsto \text{TRUE}\}$ 下，我们有 $a \models \varphi$；另一方面，在赋值 $b = \{P \mapsto \text{TRUE}, Q \mapsto \text{TRUE}\}$ 下，我们有 $b \not\models \varphi$。注意，要么 $a \models \varphi$，要么 $a \models \neg \varphi$。

命题逻辑在为硬件线路做规约时很有用，因为数字系统常基于两个值 0 和 1，它们可分别被映射为 FALSE 和 TRUE，运算符“∧”“∨”和“¬”则分别与电路的与门、或门和非门对应。

3.5 证明一阶逻辑公式

给定一个逻辑，我们可能想要证明公式 φ 对于（每个赋值和）一个给定的结构 $\mathcal{S}$ 成立，即 $\mathcal{S}$ 是 φ 的模型。在某些情况下，我们可能想要证明 φ 是重言式，即对于每个结构和该结构上的每个赋值，φ 都成立。

我们也可能需要证明一个公式在某些给定的假设下成立。形式化地说，令 Γ 是一阶公式的集合，而 φ 是一阶公式。如果 Γ 中所有的公式都成立时 φ 也成立，我们就说*在结构 $\mathcal{S}$ 下由 Γ 可推导得到 φ*。也就是，对于每个赋值 a，如果对每个 $\psi\in\Gamma$，$M_a(\psi)=\text{TRUE}$，那么 $M_a(\varphi)=\text{TRUE}$。我们称 Γ *是一组假设而 φ 是结论*，记为 $\Gamma\models^{\mathcal{S}}\varphi$。这是形式化定义的记法 $\models^{\mathcal{S}}\varphi$ 的一般化，与 $\Gamma=\varnothing$ 时的新记法一致。

当 $\Gamma\models^{\mathcal{S}}\varphi$ 在所有结构 $\mathcal{S}$ 下都成立时，我们说由 Γ *可推导得到* φ，并写作 $\Gamma\models\varphi$。例如，我们想要证明 $\{v1\equiv v2\}\models f(v1)\equiv f(v2)$。根据定义，假设和结论都在相同的赋值下解释。因此，其含义为 $v1$ 和 $v2$ 在假设和结论中都有相同的值。

我们感兴趣的是证明 $\Gamma\models\varphi$。证明的过程必须只包含简单的步骤，这些步骤的正确性应当很明显。在假设 Γ 下 φ *可证明*（provable）记作 $\Gamma\vdash\varphi$。回顾一下，φ 是重言式被记作 $\models\varphi$。相应地，$\vdash\varphi$ 表示 φ 在任何结构和赋值下都可被证明成立。

一个证明系统（proof system）包括*一组公理*（axiom）*和证明规则*（proof rule）。每个公理都是公式的*模板*（template）；模板公式可能包含*一组模板变量*（template variable）而不是子公式。公理的*实例*（instance）通过将模板变量替换为实际的子公式（或项）得到。举例来说，考虑模板公式 $\varphi\rightarrow(\psi\rightarrow\varphi)$。它包含两个模板变量：$\varphi$ 和 ψ。这些变量可以代表子公式。如果我们将 φ 替换为 $P(x)$ 并将 ψ 替换为 $Q(y)\vee P(y)$（其中 P 和 Q 是一元关系符号），我们得到 $P(x)\rightarrow((Q(y)\vee P(y))\rightarrow P(x))$。（注意在公式 $Q(y)\vee P(y)$ 中省略的最外层的括号在替换中恢复了，这么做是为了得到一个良结构的公式。）每个实例化的公理必是重言式。

一个证明规则包括一个模板公式的有限集合和另一个模板公式，分别称为*前提*（premise）和*结论*（consequent）。我们通常用这样的写法表示一个证明规则：将前提写在一条横线上面，而将结论写在下面。前提和结论往往共享一些模板变量。我们可以在证明中替换模板变量，但是要求该模板变量的每次出现都被替换为相同的子公式（或项）。替换之后，我们从前提得到一组一阶公式 Γ，并从结论得到公式 φ。为了使证明规则有效，我们必须有 $\Gamma\models\varphi$。公理也可被看做证明规则的特例，其结论不需要前提即成立。

举例来说，下面的证明规则中

$$\frac{\varphi,\varphi\rightarrow\psi}{\psi}$$

我们可将 φ 替换为 $P(x)$ 并将 ψ 替换为 $Q(y)\vee P(y)$，得到下面的实例：

$$\frac{P(x),P(x)\rightarrow(Q(y)\vee P(y))}{Q(y)\vee P(y)}$$

我们证明了 $\{P(x),\ P(x)\rightarrow(Q(y)\vee P(y))\}\models Q(y)\vee P(y)$ 的确成立。

我们接下来展示一个证明系统的例子，这个系统假设布尔运算符只包括“→”和“¬”。这样的假设就足够了，因为不难发现运算符“∧”和“∨”可消去：可以用 $(\neg\varphi)\rightarrow\psi$ 替代 $\varphi\vee\psi$，并且用 $\neg(\varphi\rightarrow\neg\psi)$ 替代 $\varphi\wedge\psi$。

这个证明系统包含如下的公理：

Ax1　$\varphi \rightarrow (\psi \rightarrow \varphi)$

Ax2　$(\varphi \rightarrow (\psi \rightarrow \eta)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \eta))$

Ax3　$((\neg \varphi \rightarrow \psi) \rightarrow ((\neg \varphi \rightarrow \neg \psi) \rightarrow \varphi))$

Ax4　$(\forall v(\varphi \rightarrow \psi)) \rightarrow (\varphi \rightarrow \forall v\psi)$，$v$ 在 φ 中不是自由的

Ax5　$(\forall v\varphi(v)) \rightarrow \varphi(e)$

在最后一个公理中，e 是一个代表项的模板变量，用来替换 φ 中 v 的所有自由出现。下面还有三个公理处理相等关系：

Ax6　$e \equiv e$

Ax7　$e_i \equiv e_i' \rightarrow f(e_1, \cdots, e_i, \cdots, e_n) \equiv f(e_1, \cdots, e_i', \cdots, e_n)$，$f \in F$

Ax8　$e_i \equiv e_i' \rightarrow (r(e_1, \cdots, e_i, \cdots, e_n) \rightarrow r(e_1, \cdots, e_i', \cdots, e_n))$，$r \in R \cup \{\equiv\}$

在这些公理之外，我们还有如下证明规则：

MP（肯定前件（Modus Ponens））：

$$\frac{\varphi, \varphi \rightarrow \psi}{\psi}$$

GEN（泛化（Generalization））：

$$\frac{\varphi}{\forall v\varphi}$$

这里，要求 v 不在证明的任何假设中自由出现。

3.5.1　正向推理

证明可以通过*正向推理*（forward reasoning）或*反向推理*（backward reasoning）的方法进行。在正向推理中，证明 $\Gamma \vdash \varphi$ 由一系列带标号的公式组成，并由公式 φ 结束。正向证明中的每一行必须满足如下情况之一：

1. 该行包含了从假设集合 Γ 取出的一个假设。

2. 该行是证明系统中的某条公理的实例化，通过将其中的模板变量替换为恰当的子公式或项而得到。

3. 按照如下的方法，使用证明系统的某条证明规则根据前面的行得到：在将证明规则中出现的模板变量实例化之后，所有的前提都是证明中前面的某些行。此次实例化后的结论成为新的一行。

我们通过证明一个简单的属性 $P(x) \rightarrow P(x)$ 来演示证明系统。

1	$P(x) \rightarrow ((P(x) \rightarrow P(x)) \rightarrow P(x))$	Ax1
2	$(P(x) \rightarrow ((P(x) \rightarrow P(x)) \rightarrow P(x))) \rightarrow$ $((P(x) \rightarrow (P(x) \rightarrow P(x))) \rightarrow (P(x) \rightarrow P(x)))$	Ax2
3	$(P(x) \rightarrow (P(x) \rightarrow P(x))) \rightarrow (P(x) \rightarrow P(x))$	MP 1,2
4	$P(x) \rightarrow (P(x) \rightarrow P(x))$	Ax1
5	$P(x) \rightarrow P(x)$	MP 3,4

第 1 行由公理 Ax1 得到，其中的 φ 被实例化为 $P(x)$ 且 ψ 被实例化为 $P(x) \rightarrow P(x)$。第 2 行由公理 Ax2 得到，其中的 φ 和 η 都被实例化为 $P(x)$，且 ψ 被实例化为 $P(x) \rightarrow P(x)$。第 3 行应用证明规则 MP 得到，其中的 φ 被实例化为 $P(x) \rightarrow ((P(x) \rightarrow P(x)) \rightarrow P(x))$，且 ψ 被实例化为 $(P(x) \rightarrow (P(x) \rightarrow P(x))) \rightarrow (P(x) \rightarrow P(x))$。经过这个实例化，MP 的第一个前提 φ 对应证明的第 1 行，而第二个前提 $\varphi \rightarrow \psi$ 对应第 2 行。第 3 行对应相同实例化下 MP 的结论。这样，第 3 行成立的理由是 MP 的应用以及前面的第 1 行和第 2 行。第 4 行再次由公理 Ax1 经实例化 φ 和 ψ 为 P x

得到。最后，第5行将MP中的φ实例化为$P(x)\rightarrow(P(x)\rightarrow P(x))$、将$\psi$实例化为$P(x)\rightarrow P(x)$。这样，MP的前提$\varphi$和$\varphi\rightarrow\psi$分别对应第4行和第3行，而第5行则对应结论。

3.5.2 反向推理

在反向推理中，我们倒着“读”每个证明规则：它描述了如何将对结论的证明任务归约为对其前提的证明任务。直观地讲，我们想要从需要证明的目标开始，然后通过使用较简单的子目标来证明这个目标。为了演示这一点，考虑下面的简单的证明规则CONJ：

$$\frac{\varphi,\psi}{\varphi\wedge\psi}$$

这个规则是说，为了证明$\varphi\wedge\psi$，我们可以分别证明φ和ψ。在正向证明推理中，我们这样使用这条规则：首先证明φ和ψ，接下来应用CONJ以得到$\varphi\wedge\psi$。在反向推理中，我们通过创建两个新的子目标φ和ψ来证明目标$\varphi\wedge\psi$。然后我们尝试分别验证这些子目标。当我们证明了子目标φ或ψ，就可以将它们从待证的子目标的列表中消除（discharge）。当这两个子目标φ和ψ都被消除了之后，目标$\varphi\wedge\psi$就被证明了，并同样可被消除。

一般来说，在反向推理中使用一个证明规则时，每个前提都对应一个子目标。这些子目标用于证明对应于规则的结论的目标。当所有的子目标都被消除时，这个目标本身也就消除了。如果某个子目标是一个公理的实例，那么它可以被立即消除。

反向推理策略往往比正向推理更直观，也更易于为进行验证的人们使用。对于那些将目标变成较短的子目标的证明规则，例如CONJ，尤其如此。前面给出的规则MP在反向推理中比CONJ要难用得多：为了证明ψ，我们需要找到某个公式φ并同时证明φ和$\varphi\rightarrow\psi$。我们将在3.8节和3.9节看到反向推理的例子。

有时，动态地改变证明中用到的假设列表将带来方便。考虑演绎定理（deduction theorem）的例子，这个定理为：

$$\Gamma\models\varphi\rightarrow\psi \quad \text{iff} \quad \Gamma\cup\{\varphi\}\models\psi$$

我们可以按照如下方法利用这条定理：为了证明$\Gamma\models\varphi\rightarrow\psi$，我们将$\varphi$作为另一个假设加入已有的假设集合中，然后证明$\Gamma\cup\{\varphi\}\models\psi$。

如果我们把公理和证明规则写成这样的形式：其中的前提和结论都形如$\Gamma\vdash\varphi$，其中$\Gamma\cup\{\varphi\}$是模板公式，我们就可以把这种灵活性引入到我们的证明系统中。例如，表示演绎定理的证明规则可记为：

$$\frac{\Gamma\cup\{\varphi\}\vdash\psi}{\Gamma\vdash\varphi\rightarrow\psi}$$

使用这个格式，我们可以将证明规则CONJ重写为：

$$\frac{\Gamma\vdash\varphi,\Gamma\vdash\psi}{\Gamma\vdash\varphi\wedge\psi}$$

这个规则说明（对应于结论的）目标和（对应于前提的）子目标的假设集合没有被改变。

练习3.5.1 用反向推理证明$P(x)\rightarrow P(x)$。

3.6 证明系统的属性

本节我们讨论关于形式化数学证明的一些基本问题，并将特别关注一阶逻辑的可证明性的问题。为说明这些讨论的动机，我们考虑著名的费马猜想（Fermat's conjecture）：

不存在自然数l，m，n，使得对于$x>2$，有$l^x+m^x=n^x$。

这个猜想在提出后的 350 年中一直都是悬而未决的，并在最近才得以证明 [130]。现在，考虑下面的计划：我们首先定义一阶签名和结构，使其包含自然数，以及费马猜想中用到的恰当的关系和函数。然后我们将费马猜想写成如下的公式：

$$\neg \exists x \exists l \exists m \exists n (x > 2 \wedge l^x + m^x \equiv n^x)$$

我们期望定义一个证明系统以证明该猜想。进而，我们期望找到一个算法以自动寻找这样的证明。而费马定理长时间悬而未决，即使是在快速的计算机问世之后依然如此，说明上面给出的计划有问题。

这里，我们对系统的一些重要的属性进行形式化并加以证明。这些属性体现了如下与形式化数学证明能力有关的本质问题。例如：

- 证明“正确”吗？
- 能否用给定的证明系统证明某个结构的所有事实？
- 是否存在一个算法能够判定给定的公式 φ 是重言式，或者可由 Γ 推导得到？$\mathcal{S}$ 是不是 φ 的模型？

3.6.1　正确性

证明系统的*正确性*（soundness）的含义是：该系统只能被用来证明正确的事实。也就是说，如果 $\Gamma \vdash \varphi$，那么一定有 $\Gamma \models \varphi$。一阶逻辑有很多不同的正确的证明系统。然而仍然需要小心谨慎，因为在将公理或证明规则形式化时可能也时常会发生错误。

例如，考虑 3.5 节给出的证明系统。假设我们忘记给出证明规则 GEN 上的约束，即全称量词不能作用于在给定前提中出现的自由变元上。那么，给定前提 $v1 \equiv v2$，我们可得到 $f(v1) \equiv f(v2)$（应用 Ax7 和 MP）。没有了对规则 GEN 的限制，我们可以应用 GEN 两次，证得 $\forall v1 \forall v2 (f(v1) \equiv f(v2))$。也就是说，我们不正确地“证明”了如下结论：如果（对于某个给定的赋值 a）$v1 \equiv v2$，那么函数 f 对任何参数都会给出常量。注意，在结论中受约束的 $v1$ 和 $v2$ 和在前提中自由出现的 $v1$ 和 $v2$ 没有任何关系。很明显，没有上述约束，这个证明系统就是不正确的。

3.6.2　完备性

证明系统的*完备性*（completeness）的含义是：如果 $\Gamma \models \varphi$，那么一定有 $\Gamma \vdash \varphi$。哥德尔（Gödel）证明了一阶逻辑是完备的。很多一阶逻辑上的证明系统是完备的，允许人们证明任何形如 $\Gamma \models \varphi$ 的正确断言。在 3.5 节我们展示了一个完备的证明系统。证明一个证明系统的完备性不是一件容易的事情，可参见其他文献，例如 [38]。

注意完备性指的是任何对于所有可能的一阶结构都成立（即解释为 TRUE）的断言，都可得到证明。然而，这并不意味着我们能够证明一个特定结构（例如我们下面要讨论的、带加法和乘法的整数结构）上的所有正确的属性。

3.6.3　可判定性

一阶逻辑是*半可判定的*（semi-decidable）。也就是说，没有算法能够检验 $\Gamma \models \varphi$ 是否成立。然而存在这样的一个算法，当 $\Gamma \models \varphi$ 成立时，能够构造出 $\Gamma \vdash \varphi$ 的证明。但是，如果 $\Gamma \not\models \varphi$，这个算法甚至不能保证会终止。只要我们能按照需要不断扩充存储容量，就可以构造得到 $\Gamma \vdash \varphi$ 的一个证明。（在图灵机的理论模型中，这不是问题，见 2.4 节。但在实际的机器上，存储是有限的，我们可能会耗尽存储。）正如在 2.4 节中解释的，用于解决半可判定问题的算法的复杂度没有上界。

3.6.4 结构完备性

给定结构$\mathcal{S}$，我们期望能够通过算法给出公式的集合$\Gamma_{\mathcal{S}}$，使得$\Gamma_{\mathcal{S}}\vdash\varphi$当且仅当$\models^{\mathcal{S}}\varphi$。将$\Gamma_{\mathcal{S}}$作为我们的完整证明系统的附加公理之后，这个证明系统就能够证明在结构$\mathcal{S}$上成立的任何属性。

对于一些一阶结构[38]，我们可以获得结构完备性。例子之一就是包括带有加法和比较关系的、但不含乘法的整数结构。这个结构被称为*布利斯博格算术*（Presburger Arithmetic）。对于这个结构，我们可以判定哪些属性成立、哪些属性不成立。布利斯博格算术上，已知的最好的判定过程的复杂度是三阶指数级的，即其执行的时间复杂度是$\mathcal{O}(2^{2^{2^{P(n)}}})$，其中$n$是公式的长度。这个问题的已知下界只有二阶指数级，即$\mathcal{O}(2^{2^{P(n)}})$，但是还没有这个复杂度的已知算法。这样就存在一个未解决的复杂度差距，并有待更多的研究[107]。

对于其他一些结构来说，就可能不存在完备的公理化方法。这包括带加法和乘法的自然数（即正整数和零）——称作*皮亚诺算术*（Peano Arithmetic）。另一个这样的结构是带有加法和乘法的整数。事实上，根据哥德尔的著名结果，这个情况并不是巧合，也不是没有投入足够的资源来寻找结构完备的公理系统：可以证明不存在这样的公理系统！这样，我们就不用期望能够证明，或判定，甚至枚举所有在这些结构上成立的一阶公式了。

即使我们知道了哥德尔不完备性定理，对于自然数或整数的（不完备的）一阶逻辑证明系统来说仍然很有用。有一些在自然数域上成立的有用的公理集合，这些公理可用来证明很多重要的属性。事实上，实践证明：自然数（整数也是这样）上的大多数重要属性，可用一阶证明系统证明。

人们常常需要在逻辑的表达能力与一些属性（如存在完备的证明系统或可判定性）之间作出权衡。例如，二阶逻辑比一阶逻辑有更强的表达能力，它可以表示图上两点之间存在一条路径这样的属性；然而，二阶逻辑没有完备的证明系统；不仅如此，二阶逻辑甚至不是半可判定的。

3.7 证明命题逻辑属性

命题逻辑是可判定的。检查是否存在赋值以满足一个给定命题逻辑公式的问题，被称为*可满足性问题*（satisfiability problem）。该问题的复杂度是NP-完备的[51]（参见第2章）。这说明，最好的已知算法的运行时间，在最坏的情况下，是公式规模的指数量级。可以用一个极其简单的算法来检查一个命题逻辑公式是否对于公式中出现的全部命题变元的所有真值成立。因为一个公式中只有有限多个变元，所以这个算法一定是可终止的。事实上，如果有n个变元，检查所有2^n个布尔赋值组合需要指数时间。

请注意，一个命题公式φ是重言式当且仅当$\neg\varphi$不可满足（也就是没有赋值能够使得$\neg\varphi$计算为TRUE）。这样，要说明φ不是重言式，只需要找到一个满足$\neg\varphi$的赋值就足够了。已经有一些有效的启发式方法用于检查可满足性，例如，SATO[151]、GRASP[129]和Stalmark方法[136]。

可以给出一个相对完备的证明系统来证明命题逻辑的重言式。一个例子就是希尔伯特系统（Hilbert's system），它包括了3.5节的公理**Ax1**～**Ax3**和证明规则**MP**。这些公理和证明规则中出现的变量可由任意命题公式替代。我们假设所有命题公式都首先经过转换，使之仅包含否定（$\neg$）和蕴含（$\rightarrow$）运算符。然而，因为存在适当的检验算法，在手工或计算机辅助证明中很少会直接使用命题逻辑的证明规则。

3.8 一个实用的证明系统

我们现在介绍一个用于一阶逻辑的简单证明系统，这个系统和 PVS 系统 [109] 类似。一个*相继式*（sequent）是如下形式的公式：

$$(\varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n) \rightarrow (\psi_1 \vee \psi_2 \vee \cdots \vee \psi_m)$$

一个相继式的每个子公式 φ_i 都是该相继式的*前件*（antecedent），每个子公式 ψ_i 都是该相继式的*后继*（succedent）。公式 φ_i，…，φ_n 和 ψ_1，…，ψ_m 不限定于任何一个特定形式。因为我们总是可以令 $n=0$ 而 $m=1$，所以每个一阶公式实际上都是这种形式。我们现在展示一个利用了相继式形式的证明系统。

我们将公式竖着写，其中在横线之上是前件，每行写一个，并用负数（−1、−2 等）编号；后继写在横线下方，用正数（1、2 等）编号。

$$\begin{array}{cc} -1 & \varphi_1 \\ -2 & \varphi_2 \\ \vdots & \vdots \\ -n & \varphi_n \\ \hline 1 & \psi_1 \\ 2 & \psi_2 \\ \vdots & \vdots \\ m & \psi_m \end{array}$$

在一个证明中，为了方便起见，我们将证明过程中最近修改的几行标记为“*”。

开始时，证明 $\Gamma \vdash \psi$ 是一个没有前提，只有一个后继 ψ 的相继式，这个相继式称为*初始目标*。这个目标现在正等待证明。这样，待证目标列表一开始仅包含 ψ。我们可以用一条公理将一个目标从待证目标列表中消除，或用一条证明规则来将其归约为一些更简单的子目标，这些子目标又被加入待证目标的列表。如果我们能够将前面的某个目标产生的所有待证子目标全部消除，那么这个目标本身也就可以消除了。因此，我们的目标就是不断重复这个过程，直到可以消除初始目标。

在 2.3 节描述的树可用来描述当前的证明状态。一棵*证明树*的根是初始目标 ψ。树中任何节点的直接后继，都是在该节点对应的目标上应用某个证明规则而产生的子目标。这棵树的某个叶子节点是*当前目标*。在证明的过程中，我们可能会找到一个证明规则，将这个目标归约为一些子目标。这些新的子目标将作为当前目标的直接后继加入树中，使得树逐步增长。我们也可能通过使用恰当的公理来消除当前目标。消除一个节点的所有后继（即子目标）之后就可以消除该节点本身。证明过程可看做是一系列树之间的变形（tree transformation，参见图 3.1）。我们的目标即完全缩减这棵树，最终得到一棵空树（即没有节点的树）。我们将在 3.9 节看到使用树变形的证明示例。

这里将给出的证明规则都来自 PVS 系统。PVS 系统基于一种比一阶逻辑更通用的逻辑，这种*高阶逻辑*以*类型论*（type theory）[27] 为基础，这里我们不再对其进一步讨论，我们只讨论这个证明系统中能够处理一阶逻辑的证明规则子集。这里，我们不直接展示证明规则本身，而是给出 PVS 命令，这些命令能控制它的机械化定理证明器。每个命令都是基于一个简单证明规则或者一条公理的。（一个有趣并且不太难的练习就是写出每个命令对应的证明规则。）

消去（eliminate） 如果下面的条件之一满足，则去除当前的目标，即如下形式的相继式 $(\varphi_1 \wedge \cdots \wedge \varphi_n) \rightarrow (\psi_1 \vee \cdots \vee \psi_m)$：

- 前件 φ_i，…，φ_n 中的某个公式为 *false*。

- 后继 ψ_1，…，ψ_m 中的某个公式为 *true*。
- 某个前件和其中一个后继相同，即对于某个 $1\leqslant i\leqslant n$ 以及 $1\leqslant j\leqslant m$，我们有 $\varphi_i=\psi_j$。
- 其中一个后继形如 $e\equiv e$，其中 e 是一个项。

消去命令对应于应用一条公理（因为它去除了一个目标）。事实上，在 PVS 系统中，每当构造出一个新目标时，这个命令就会被自动应用。

扁平化（flatten） 基于相继式的特殊形式，通过简化当前的目标构造得到一个直接子目标。这个命令试图通过如下所示的一条或多条化简方法，产生一个带有较短子公式的相继式：

- 否定去除。去除否定的一个方法是将一个形如 $\neg\psi$ 的前件替换为形如 ψ 的后继。这个方法基于 $(\varphi\wedge\neg\psi)\rightarrow\eta$ 和 $\varphi\rightarrow(\psi\vee\eta)$ 逻辑等价的事实。另一种方法是，将形如 $\neg\psi$ 的后继替换为形如 ψ 的前件。这个方法基于 $\varphi\rightarrow(\neg\psi\vee\eta)$ 和 $(\varphi\wedge\psi)\rightarrow\eta$ 逻辑等价的事实。
- 将一个形如 $\varphi_1\wedge\varphi_2$ 的前件替换为形如 φ_1 和 φ_2 的两个前件，以拆开合取式。
- 将一个形如 $\varphi_1\vee\varphi_2$ 的后继替换为形如 φ_1 和 φ_2 的两个后继，以拆开析取式。
- 将一个形如 $\varphi_1\rightarrow\varphi_2$ 的后继替换为一个前件 φ_1 和一个后继 φ_2。这一方法是基于 $\varphi_1\rightarrow\varphi_2$ 和 $(\neg\varphi_1)\vee\varphi_2$ 是相同的事实。这个过程是上面两个步骤的组合：拆开析取与否定去除。

扁平化命令指定了需要进行扁平化操作的当前目标的行号。和其他命令相似，如果扁平化失败，即指定的行不是上述情况之一，那么当前目标保持不变。

拆分（split） 根据如下几种情况之一，拆分子目标：

- 如果当前目标 s 带有形如 $\varphi_1\vee\varphi_2\vee\cdots\vee\varphi_l$ 的前件，则这个规则可以产生 l 个子目标，其中第 i 个新的子目标是将当前目标中上述前件替换为 φ_i 而得到的。
- 如果当前目标 s 含有形如 $\varphi_1\wedge\varphi_2\wedge\cdots\wedge\varphi_l$ 的后继，则这个规则可产生 l 个子目标，其中第 i 个新的子目标是将当前目标中的上述后继替换为 φ_i 而得到的。

拆分命令指定了需要进行拆分操作的当前目标的行号。

假设（assumption） 选择一个子目标，并将 Γ 中的某个公式作为一个新的前件加入，产生一个直接子目标。假设命令指定了在 Γ 中加入的公式的名字。

斯科伦化（Skolemize） 删除后继的外层全称量词，或前件的外层存在量词。有如下两种可能：

- 存在一个形如 $\forall v\psi$ 的后继。斯科伦化用 $\psi[c/v]$ 替换该后继，产生一个新的子目标。其中 c 是一个新的常量，且没有在之前的证明中出现。使用这个规则的理由如下：如果新的子目标得以证明，因为 c 的选择不失一般性（因为在选择该常量的时候它未被使用过并且没有任何特别的约束，所以在其上没有任何限制），所以我们可以推断带有全称量词的原有子目标成立。
- 存在一个形如 $\exists v\varphi$ 的前件。斯科伦化用 $\psi[c/v]$ 替换该前件，产生一个新的子目标。其中 c 是一个新的常量，且没有在之前的证明中出现。如果新的子目标得以证明，我们可以说除了要求 c 满足 φ 之外没有对其做任何限制。这样（不失一般性），满足 φ 的值的存在性可以证明这个蕴含式，得到带有存在量词的原有子目标。

斯科伦化命令指定了需要执行斯科伦化的行以及常量名 c。

实例化（instantiate） 从某种意义上说，实例化是斯科伦化的补，因为它允许去除一个前件中的最外层全称量词，或一个后继中的最外层存在量词。它也有以下两种情况：

- 存在一个形如 $\forall v\varphi$ 的前件。那么我们可以任取一个项 e，并通过将该前件替换为 $\varphi[e/v]$ 产生一个新的子任务。如果我们之后能够证明当我们要求 φ 在特定项 e 上成立时，蕴含式成立，那么很明显，当我们要求 φ 在论域中所有值上都成立时，该蕴含式成立。
- 存在一个形如 $\exists v\psi$ 的后继。那么我们可以任取一个项 e，通过将该后继替换为 $\psi[e/v]$ 产

生一个新的子任务。使用这个规则的理由是：如果蕴含式在 ψ 满足项 e 的特定值时成立，那么我们知道，存在一个值使得 ψ 成立。也就是说我们用项 e 作为（当其前件成立时）ψ 成立的证据。

注意，在上面的两种情况中，当我们将一个自由变元替换为一个表达式的时候，替换的定义（见 3.3 节）不允许该表达式包含了在替换处受限的变元。*实例化*命令带有需要执行该命令的行以及项 e。注意斯科伦化不如实例化自由，它仅允许用新的变元进行替换。

替换（substitution）　给定一个形如 $e_1 \equiv e_2$ 的前件，我们可以产生一个新的子目标，在前件或者后继中，将 e_1 替换为 e_2，或将 e_2 替换为 e_1。这个命令表明等于符号“≡”的语义就是通常的等价。

替换命令指定了获得 $e_1 \equiv e_2$ 的行，需要应用替换命令的行，以及需要将 e_1 替换为 e_2 还是反过来。

证明系统选择证明树上的某个叶节点作为当前目标。我们可以通过命令*推迟*（postpone）改变这个选择，该命令循环地在当前叶节点集合中进行遍历。其效果是选择一个新的叶节点作为当前目标（如果当前树有多个叶节点的话），将证明当前叶节点的过程推迟到稍后的证明中。

3.9　证明示例

在第一个例子中，我们证明下面的命题逻辑重言式：

$$((A \to C) \wedge (B \to C)) \to ((A \vee B) \to C)$$

这样，我们的初始目标（编号为 1）如下：

$$1 \quad ((A \to C) \wedge (B \to C)) \to ((A \vee B) \to C)$$

初始证明树，即图 3.1 中左上方的树 1，包含这个唯一的节点。注意初始目标的后继形如 $\varphi \to \psi$，其中 $\varphi = (A \to C) \wedge (B \to C)$ 并且 $\psi = ((A \vee B) \to C)$。因此，对行 1 应用扁平化，我们得到初始目标的子目标如下：

$$\begin{array}{lll} * & -1 & (A \to C) \wedge (B \to C) \\ \hline * & 1 & (A \vee B) \to C \end{array}$$

当前的证明树，即图 3.1 中的树 2，包含子目标节点 2，这个节点是树的唯一叶节点，并且是编号为 1 的初始目标的直接后继。

在行 −1 上应用扁平化，我们得到如下的子目标，编号为 3。

$$\begin{array}{lll} * & -1 & A \to C \\ * & -2 & B \to C \\ \hline & 1 & ((A \vee B) \to C) \end{array}$$

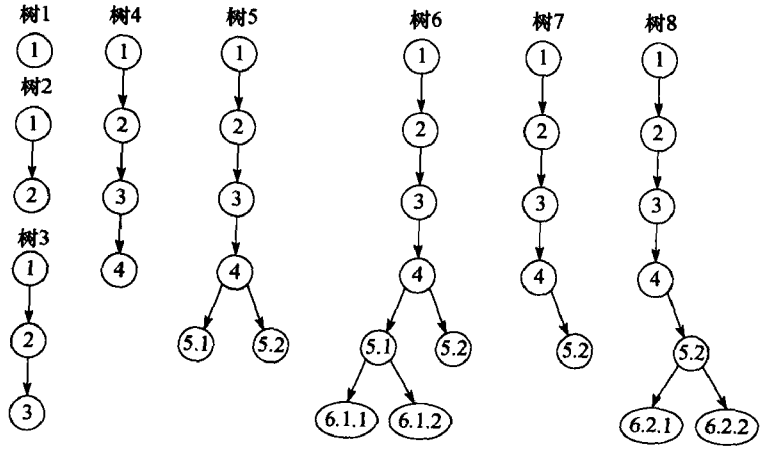


图 3.1　证明树

当前的证明树，即树3，包含一个带有3个节点的路径，其中的新叶节点是当前子目标。在行1上应用扁平化，我们获得如下的子目标，编号为4。

$$\begin{array}{cll} & -1 & A \rightarrow C \\ & -2 & B \rightarrow C \\ * & -3 & A \vee B \\ \hline * & 1 & C \end{array}$$

这样做向证明树中加入了新的叶节点，得到树4。我们不能再继续用扁平化来简化节点4中新的当前目标了。但是我们能在行−3上应用拆分，获得下面两个子目标：

$$\begin{array}{cll} & -1 & A \rightarrow C \\ & -2 & B \rightarrow C \\ * & -3 & A \\ \hline & 1 & C \end{array} \qquad \begin{array}{cll} & -1 & A \rightarrow C \\ & -2 & B \rightarrow C \\ * & -3 & B \\ \hline & 1 & C \end{array}$$

这两个子目标是节点4的直接后继，在新的证明树即树5上标记为5.1和5.2。我们需要分开考虑这两个子目标。我们首先把左边的子目标，即节点5.1，作为新的当前目标。我们根据行−1拆分。请注意，$A \rightarrow C$ 和 $(\neg A) \vee C$ 逻辑等价，因此拆分的结果或是把 A 的否定形式加为新的后继，或是将 C 加为新的前件。这样得到的两个子目标是：

$$\begin{array}{cll} & -1 & B \rightarrow C \\ & -2 & A \\ \hline & 1 & C \\ * & 2 & A \end{array} \qquad \begin{array}{cll} * & -1 & C \\ & -2 & B \rightarrow C \\ & -3 & A \\ \hline & 1 & C \end{array}$$

在新的证明树即树6上，节点5.1的这些子目标被编号为6.1.1和6.1.2。注意我们暂时没有考虑叶节点5.2。我们后面必须处理这个节点，以完成证明。在6.1.1和6.1.2两个节点中，都有一个子公式既是前件又是后继。这样，这些节点可以通过消去命令以某种（任意的）顺序被消除。这两个命令使得6.1.1和6.1.2两个节点的共同父节点5.1的子目标被消除。消除后得到证明树7。这样，从树6到树7的转换包含了三步：(1) 消除节点6.1；(2) 消除节点6.2；(3) 消除节点5.1。

新的证明树即树7的唯一叶节点是节点5.2。因此，它成为当前目标。回顾一下，这个节点包含如下的相继式：

$$\begin{array}{cll} & -1 & A \rightarrow C \\ & -2 & B \rightarrow C \\ * & -3 & B \\ \hline & 1 & C \end{array}$$

我们现在可以根据行−2通过拆分处理这个新的当前目标。我们得到下面两个子目标：

$$\begin{array}{cll} & -1 & A \rightarrow C \\ * & -2 & C \\ & -3 & A \\ \hline & 1 & C \end{array} \qquad \begin{array}{cll} & -1 & A \rightarrow C \\ & -2 & B \\ \hline & 1 & C \\ * & 2 & B \end{array}$$

这两个子目标，6.2.1和6.2.2，是树8的新的叶节点。因为它们分别包含相同的前件和后继，所以这两个节点仍然可以通过消去命令被消除。现在，消除过程可以一直扩展到证明树的根节点：当其前驱6.2.1和6.2.2被消除之后，节点5.1被消除。出于同样的原因，节点4被消除，接着按照创建它们的相反顺序，节点3、2和1依次被消除。当根节点消除后，证明完成。

练习 3.9.1 试证明：

- $(A \wedge (\neg A)) \rightarrow B$。
- 给定前提 $A \rightarrow B$ 和 $B \rightarrow C$，我们可以得到 $A \rightarrow C$。

举另一个更为复杂的例子，我们将证明群论的一个简单的属性。这次，证明过程将使用一阶逻辑。证明将展示实例化和斯科伦化的运用。群论的签名包含群乘法符号“×”（这是用于实数或整数乘法的相同数学符号的另一个重载），以及单位元 *unit*。一个群满足如下的三个公理：

单位元律（unity）　存在一个特别的元素 *unit*，使得对于每个 x 都有 $x \times unit \equiv x$。

$$\forall x x \times unit \equiv x$$

结合律（associativity）　运算符×满足结合律，也就是说在重复应用这个运算符时，括号的顺序和结果无关。

$$\forall x \forall y \forall z (x \times y) \times z \equiv x \times (y \times z)$$

右逆元律（right-complement）　每个元素 x 都有一个*逆元*（complement）y 使得 $x \times y$ 等于单位元 *unit*。

$$\forall x \exists y (x \times y \equiv unit)$$

注意 y 是 x 的*右*逆元。

我们要证明对于每个元素 x，都存在一个*左*逆元 y 使得 $y \times x$ 等于单位元 *unit*。形式化地写作，$\forall x \exists y (y \times x \equiv unit)$。我们首先给出非形式化的证明，正如在数学课或代数书上所做的那样。

首先，我们任意选择 x。因为我们不限制 x，我们的证明对 x 的每一个可能值都成立（或者说“不失一般性”）。根据“右逆元律”公理，存在一个值 y 使得

$$x \times y \equiv unit \tag{3.3}$$

再次使用“右逆元律”公理，存在一个值 z 使得

$$y \times z \equiv unit \tag{3.4}$$

我们用下面的等式证明了这个属性：

	$y \times x$	
$\equiv$	$(y \times x) \times unit$	单位元律
$\equiv$	$(y \times x) \times (y \times z)$	式(3.4)
$\equiv$	$y \times (x \times (y \times z))$	结合律
$\equiv$	$y \times ((x \times y) \times z)$	结合律
$\equiv$	$y \times (unit \times z)$	式(3.3)
$\equiv$	$(y \times unit) \times z$	结合律
$\equiv$	$y \times z$	单位元律
$\equiv$	$unit$	式(3.4)

通过使用 PVS 证明系统得到的形式化证明（为便于表示，这里稍作改动），和上面的非形式化证明的思路一样。然而，形式化证明则以更为细致和严格的方式完成。这个证明是“线性的”，因为当前的证明树总是每次增加恰好一个新节点，这个节点就是新的子目标。在证明的最后一步，最后一个子目标被消除，从而使得证明树上的所有节点都按照其创建顺序的逆序被消除。

我们由如下的初始目标开始：

$$\frac{}{1 \quad \forall x \exists y y \times x \equiv unit}$$

这样，我们得到仅包含一个节点的树，该节点即当前子目标。将后继 1 中的变量 x 斯科伦化为新的常量 x'，我们得到一个新的子目标

$$\frac{}{* \quad 1 \quad \exists y y \times x' \equiv unit}$$

应用“右逆元律”假设

$$\frac{* \quad -1 \quad \forall x \exists y x \times y \equiv unit}{1 \quad \exists y y \times x' \equiv unit}$$

将前件－1 中的变量 x 实例化为常量 x'。

$$\frac{\begin{array}{lll} * & -1 & \exists y x' \times y \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

将前件－1 中的变量 y 斯科伦化为新常量 y'。

$$\frac{\begin{array}{lll} * & -1 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

应用“右逆元律”假设

$$\frac{\begin{array}{lll} * & -1 & \forall x \exists y x \times y \equiv unit \\ & -2 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

前件－1 中的变量 x 实例化为常量 y'。

$$\frac{\begin{array}{lll} * & -1 & \exists y y' \times y \equiv unit \\ & -2 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

前件－1 中的变量 y 斯科伦化为新的常量 z'。

$$\frac{\begin{array}{lll} * & -1 & y' \times z' \equiv unit \\ & -2 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

应用“单位元律”假设

$$\frac{\begin{array}{lll} * & -1 & \forall x x \times unit \equiv x \\ & -2 & y' \times z' \equiv unit \\ & -3 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

前件－1 中的变量 x 实例化为项 $(y' \times x')$。

$$\frac{\begin{array}{lll} * & -1 & (y' \times x') \times unit \equiv (y' \times x') \\ & -2 & y' \times z' \equiv unit \\ & -3 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

根据前件－2，将前件－1 中的 $unit$ 替换为 $(y' \times z')$。

$$\frac{\begin{array}{lll} * & -1 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\ & -2 & y' \times z' \equiv unit \\ & -3 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

应用“结合律”假设

$$\frac{\begin{array}{lll} * & -1 & \forall x \forall y \forall z (x \times y) \times z \equiv x \times (y \times z) \\ & -2 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\ & -3 & y' \times z' \equiv unit \\ & -4 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

前件－1 中的 x 实例化为 y'。

$$\frac{\begin{array}{lll} * & -1 & \forall y \forall z (y' \times y) \times z \equiv y' \times (y \times z) \\ & -2 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\ & -3 & y' \times z' \equiv unit \\ & -4 & x' \times y' \equiv unit \end{array}}{\begin{array}{lll} & 1 & \exists y y \times x' \equiv unit \end{array}}$$

前件－1 中的 y 实例化为 x'。

$$
\begin{array}{lll}
* & -1 & \forall z(y' \times x') \times z \equiv y' \times (x' \times z) \\
 & -2 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -3 & y' \times z' \equiv unit \\
 & -4 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists y y \times x' \equiv unit
\end{array}
$$

前件－1 中的 z 实例化为 $(y' \times z')$。

$$
\begin{array}{lll}
* & -1 & (y' \times x') \times (y' \times z') \equiv y' \times (x' \times (y' \times z')) \\
 & -2 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -3 & y' \times z' \equiv unit \\
 & -4 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists y y \times x' \equiv unit
\end{array}
$$

根据前件－2，将前件－1 中的 $(y' \times x') \times (y' \times z')$ 替换为 $(y' \times x')$。

$$
\begin{array}{lll}
* & -1 & (y' \times x') \equiv y' \times (x' \times (y' \times z')) \\
 & -2 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -3 & y' \times z' \equiv unit \\
 & -4 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists y y \times x' \equiv unit
\end{array}
$$

应用“结合律”假设

$$
\begin{array}{lll}
* & -1 & \forall x \forall y \forall z(x \times y) \times z \equiv x \times (y \times z) \\
 & -2 & (y' \times x') \equiv y' \times (x' \times (y' \times z')) \\
 & -3 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -4 & y' \times z' \equiv unit \\
 & -5 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists y y \times x' \equiv unit
\end{array}
$$

将前件－1 中的 x 实例化为 x'、y 实例化为 y' 以及 z 实例化为 z'（我们将这 3 个简单的步骤合为一个）。

$$
\begin{array}{lll}
* & -1 & (x' \times y') \times z' \equiv x' \times (y' \times z') \\
 & -2 & (y' \times x') \equiv y' \times (x' \times (y' \times z')) \\
 & -3 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -4 & y' \times z' \equiv unit \\
 & -5 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists y y \times x' \equiv unit
\end{array}
$$

使用前件－1，将前件－2 中的 $(x' \times (y' \times z'))$ 替换为 $((x' \times y') \times z')$。

$$
\begin{array}{lll}
 & -1 & (x' \times y') \times z' \equiv x' \times (y' \times z') \\
* & -2 & (y' \times x') \equiv y' \times ((x' \times y') \times z') \\
 & -3 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -4 & y' \times z' \equiv unit \\
 & -5 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists y y \times x' \equiv unit
\end{array}
$$

使用前件－5，用 $(x' \times y')$ 替换前件－2 中的 $unit$。

$$
\begin{array}{lll}
 & -1 & (x' \times y') \times z' \equiv x' \times (y' \times z') \\
* & -2 & (y' \times x') \equiv (y' \times (unit \times z')) \\
 & -3 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -4 & y' \times z' \equiv unit \\
 & -5 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists y y \times x' \equiv unit
\end{array}
$$

应用“结合律”假设

$$
\begin{array}{rll}
* & -1 & \forall x \forall y \forall z (x \times y) \times z = x \times (y \times z) \\
& -2 & (x' \times y') \times z' = x' \times (y' \times z') \\
& -3 & (y' \times x') = (y' \times (unit \times z')) \\
& -4 & (y' \times x') \times (y' \times z') = (y' \times x') \\
& -5 & y' \times z' = unit \\
& -6 & x' \times y' = unit \\
\hline
& 1 & \exists y y \times x' = unit
\end{array}
$$

将前件－1 中的 x 实例化为 y'、y 实例化为 $unit$ 以及 z 实例化为 z'。

$$
\begin{array}{rll}
* & -1 & (y' \times unit) \times z' = y' \times (unit \times z') \\
& -2 & (x' \times y') \times z' = x' \times (y' \times z') \\
& -3 & (y' \times x') = (y' \times (unit \times z')) \\
& -4 & (y' \times x') \times (y' \times z') = (y' \times x') \\
& -5 & y' \times z' = unit \\
& -6 & x' \times y' = unit \\
\hline
& 1 & \exists y y \times x' = unit
\end{array}
$$

使用前件－1，将前件－3 中的 $y' \times (unit \times z')$ 替换为 $(y' \times unit) \times z'$。

$$
\begin{array}{rll}
& -1 & (y' \times unit) \times z' = y' \times (unit \times z') \\
& -2 & (x' \times y') \times z' = x' \times (y' \times z') \\
* & -3 & (y' \times x') = ((y' \times unit) \times z') \\
& -4 & (y' \times x') \times (y' \times z') = (y' \times x') \\
& -5 & y' \times z' = unit \\
& -6 & x' \times y' = unit \\
\hline
& 1 & \exists y y \times x' = unit
\end{array}
$$

应用“单位元律”假设

$$
\begin{array}{rll}
* & -1 & \forall x x \times unit = x \\
& -2 & (y' \times unit) \times z' = y' \times (unit \times z') \\
& -3 & (x' \times y') \times z' = x' \times (y' \times z') \\
& -4 & (y' \times x') = ((y' \times unit) \times z') \\
& -5 & (y' \times x') \times (y' \times z') = (y' \times x') \\
& -6 & y' \times z' = unit \\
& -7 & x' \times y' = unit \\
\hline
& 1 & \exists y y \times x' = unit
\end{array}
$$

将前件－1 中的 x 实例化为 y'。

$$
\begin{array}{rll}
* & -1 & y' \times unit = y' \\
& -2 & (y' \times unit) \times z' = y' \times (unit \times z') \\
& -3 & (x' \times y') \times z' = x' \times (y' \times z') \\
& -4 & (y' \times x') = ((y' \times unit) \times z') \\
& -5 & (y' \times x') \times (y' \times z') = (y' \times x') \\
& -6 & y' \times z' = unit \\
& -7 & x' \times y' = unit \\
\hline
& 1 & \exists y y \times x' = unit
\end{array}
$$

使用前件－1，将前件－4 中的 $y' \times unit$ 替换为 y'。

$$
\begin{array}{cll}
 & -1 & y' \times unit \equiv y' \\
 & -2 & (y' \times unit) \times z' \equiv y' \times (unit \times z') \\
 & -3 & (x' \times y') \times z' \equiv x' \times (y' \times z') \\
* & -4 & (y' \times x') \equiv (y' \times z') \\
 & -5 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -6 & y' \times z' \equiv unit \\
 & -7 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists yy \times x' \equiv unit
\end{array}
$$

使用前件－6，将前件－4 中的 $y' \times z'$ 替换为 $unit$。

$$
\begin{array}{cll}
 & -1 & y' \times unit \equiv y' \\
 & -2 & (y' \times unit) \times z' \equiv y' \times (unit \times z') \\
 & -3 & (x' \times y') \times z' \equiv x' \times (y' \times z') \\
* & -4 & y' \times x' \equiv unit \\
 & -5 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -6 & y' \times z' \equiv unit \\
 & -7 & x' \times y' \equiv unit \\
\hline
 & 1 & \exists yy \times x' \equiv unit
\end{array}
$$

将后继 1 中的 y 实例化为 y'。

$$
\begin{array}{cll}
 & -1 & y' \times unit \equiv y' \\
 & -2 & (y' \times unit) \times z' \equiv y' \times (unit \times z') \\
 & -3 & (x' \times y') \times z' \equiv x' \times (y' \times z') \\
 & -4 & y' \times x' \equiv unit \\
 & -5 & (y' \times x') \times (y' \times z') \equiv (y' \times x') \\
 & -6 & y' \times z' \equiv unit \\
 & -7 & x' \times y' \equiv unit \\
\hline
* & 1 & y' \times x' \equiv unit
\end{array}
$$

因为前件－4 和后继 1 是相同的，我们现在可以应用消去命令将最后一个目标消除。因为证明的结构是线性的（每个目标有且仅有一个子目标），所以整个证明树（这个例子中仅有一条路径）都可被消除了，证明完成。

3.10　机器辅助证明

因为不可判定性和结构不完备性（在 3.6 节中已讨论过）的原因，数学证明需要受过相关教育的人的精妙技巧。然而，称为*机械化定理证明器*（mechanized theorem prover）的计算机系统，可以有效地帮助人们获得和检查证明。获得证明的过程往往是证明者和机械化定理证明器交互合作的结果，这里的证明者可能是数学家、逻辑学家或者受过训练的工程师。

定理证明器帮助证明者的方法之一是保证证明的严密性。一个证明包含很多公式，包括公理，并通过证明规则以精确的方式连接起来。人们在证明时往往容易犯错误。只有在证明上花了一些时间之后，证明者才会在看起来明显或者简单的地方走一些“捷径”。使用机械化定理证明器时不允许这样做，并保证每个中间步骤都严格地按照所用证明系统的规则获得。

尽管所有的定理证明器都强制要求严密性，但定理证明研究团体在严密的程度上却分为两派。极端之一是“纯粹”方法的支持者。他们认为在定理证明器使用的所有数学结论都必须要验证，且验证必须在很少量的基本公理和证明规则上进行。按照这个方法，举例来说，若是一个人想要证明字符串上的一个属性，而之前在这个论域中没有使用过该定理证明器，证明者就必须首先将字符串论域映射到定理证明器已经存在于其中的某个论域上。证明者不能新增字符串上的公理，而必须先用证明器，依据已经证明过的定理，将这些关于字符串上的新公理作为定理进

行证明。只有在此之后，才能将这些定理用于其他证明。

纯粹方法的目的明显是为了增强证明正确的可靠性。事实上，定理证明器的目的就是提供足够的可信度。在纯粹方法下，整个证明都基于非常小数量的已经广为接受的公理和证明规则，其中不大可能存在矛盾之处（存在矛盾的可能性不会比已知的、基于集合论和类型论的数学问题更大 [27]。）为了避免在使用这种系统时可能出现令人痛苦的时间开销，定理证明器整合了一些不同的已经证明过的数学理论，并将其以定理库的方式组织起来，以便在将来的证明中重复使用。这类系统的一个例子就是 HOL [97]，HOL 是高阶逻辑（higher order logic）的英文首字母缩写。

在另一个极端，“实用”方法关注于如何帮助用户更快地得到证明。它允许用户提供新论域上的公理系统，并在构建证明时立即使用它们。保证这些公理确实反映了目标论域上的属性，则是用户的任务。用户在最小的时间投入之后就可以开始使用机械化定理证明器。这个时间投入可能包括，在书中寻找给定论域上的合适的公理，并将它们转变为定理证明所接受的语法。

这个方法在工程应用上往往很有效率。然而，用户必须注意其中的风险。一个常见的现象是只要证明者认为加入更多的公理可以使证明更加简单，他们就倾向于在验证的过程中加入这些公理。证明者有可能错误地认为某个属性在给定论域上是正确的。这种“想当然”的做法可能会使得证明比预想的要弱很多：加入的公理可能实际上只是假设而已，它们可能成立也可能不成立。因此，证明者仅仅证明了属性 φ 在某个在证明过程中加入的假设之下成立，而不是证明了属性 φ 在某个论域上成立。最糟糕的情况是，如果加入的假设带有一些内部的矛盾，证明属性 φ 成立就没有任何价值了。应用“实用”方法建立的证明系统的例子之一就是 PVS [109]。本章中的证明示例和这个工具的证明过程很接近。

定理证明器能够帮助证明者的另一个方式是，在给定点上给出如何继续证明的建议。尽管机械化的定理证明器可能不能自己完成证明，但它可以给出如何继续证明的各种有用的建议。尽管对很多重要的数学结构，定理证明不能自动完成，但还是有很多启发式方法可以帮助用户决定如何继续证明。

一个策略（tactic）是一个组合（可能是有条件的或重复的）了多个公理和证明规则的应用的小过程。这些策略允许证明者在证明定理的过程中完成有效的进展，而不需要每次只应用一个公理或证明规则。这里，纯粹方法和实用方法又出现了区别。纯粹的方法允许把系统中已经存在的公理和证明规则组合起来得到策略；实用的方法通常允许这些策略作为外部程序实现，在证明中应用于某些子目标而产生新的子目标。

机械化定理证明器在证明导航中很有帮助。它们可以用来跟踪不同的未完成的证明选项，并在不同的未证明子目标之间切换，并记录完整的证明以用于后续的更新，或用于另一个证明。很多定理证明器都包含一个重要的功能，即证明的“美观显示”。一旦证明完成，这个功能可产生易读的证明文本。这些文本甚至可被用于文章或书中。

练习 3.10.1 等价关系（equivalence relation）常和形式化方法一起使用，其应用将在后续章节中深入讨论。回顾一下，在 2.1 节中，某个论域上的等价关系“$\cong$”满足下面的三个公理：

自反性 $\forall x\, x \cong x$。

对称性 $\forall x \forall y (x \cong y \rightarrow y \cong x)$。

传递性 $\forall x \forall y \forall z ((x \cong y \wedge y \cong z) \rightarrow x \cong z)$。

应用这些公理非形式化地证明下面的属性：

如果 x 和 y 不等价，那么任何与 x 等价的值 z 一定与 y 不等价。

使用一个机械化定理证明器（例如，从下面列出的 URL 中获取一个），以形式化地证明上

面的属性。形式化地说，即证明

$$\forall x \forall y((\neg x \cong y) \rightarrow \forall z(x \cong z \rightarrow \neg y \cong z))$$

3.11　机械化定理证明器

ACL2 系统由德克萨斯大学奥斯丁分校提供：

http://www.cs.utexas.edu/users/moore/acl2

Coq 系统由 INRIA 提供：

http://pauillac.inria.fr/coq

HOL 系统由剑桥大学提供：

http://www.cl.cam.ac.uk/Research/HVG/HOL

Isabelle 系统［106］由剑桥大学提供：

http://www.cl.cam.ac.uk/Research/HVG/Isabelle

Larch 系统由 MIT 提供：

http://larch.lcs.mit.edu

Nuprl 系统由康奈尔大学提供：

http://www.cs.cornell.edu/Info/Projects/NuPrl

PVS 系统由斯坦福研究所（SRI）提供：

http://pvs.csl.sri.com

TPS 系统由卡耐基-梅隆大学提供：

http://www.cs.cmu.edu/andrews/tps.html

注意：使用形式化方法系统时常常需要填写并发送适当的发布表格，以表示遵守特定的使用规范。

3.12　扩展阅读

下面的书籍可以作为数理逻辑的优秀入门读物：

G. S. Boolos, D. J. Richard, *Computability and Logic*, Cambridge University Press, (3rd edition), 1989.

D. van Dalen, *Logic and Structure*, Springer-Verlag, 3rd edition, 1994.

H. D. Ebbinghaus, J. Flum, W. Thomas, *Mathematical Logic*, Springer-Verlag, 1994.

下面的几本书籍介绍特定的演绎定理证明器：

R. S. Boyer, J. S. Moore, *The Computational Logic Handbook*, Academic Press, 1998.

T. F. Melham, M. J. C. Gordon, *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.

L. C. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge, 1990.

N. Shankar, *Mathematics, Machines and Godel's Proof*, Cambridge, 1997.

L. Wos, *The Automatic Reasoning: An Experimenter's Notebook with Otter, Tutorial*, Academic Press, 1996.

L. Wos, *Automated Reasoning: Introduction and Application*, 2nd edition, McGraw-Hill, 1992.

第4章
Software Reliability Methods

软件系统建模

“别人觉得你是怎么个人，你就是怎么个人。”——或者，如果你喜欢说得简单些，就是：“不要想象你自己不是别人心目中认为你是的那种人，你过去是怎么个人或者可能是怎么个人也并非不是更早以前他们认为你不是的那种人。”

刘易斯·卡洛尔《爱丽丝漫游奇境记》

使用工具或技术来提高软件系统的可靠性，通常需要先对其建模。也就是说，使用展现系统可观测属性的数学对象来表示该系统。在物理学中，我们常常会使用一个系统（例如，一个原子，一个行星）的一种模型来分析系统的某些属性，这种方法是很有帮助的。建模通常包含了抽象的过程，即简化系统的描述，同时仅保留有限数量的原始细节。这使得人们可以仅关注主要的属性，并且更好地控制系统的复杂度。与物理学一样，在软件中，通常处理一个抽象模型更加方便和易控制，因为相对于整个系统，抽象模型更加简单且理想化。系统的某些特性，诸如高复杂性等常常会让人们无法直接分析软件代码，而模型通常可以被构造得足够小且简单，因而可以应用形式化方法。从另一个角度来说，由于数学方法无法直接处理物理实体（例如计算机内存）但却可以对这些实体进行数学抽象，因此，建模是很有必要的。

为软件建模的另一个原因是软件的多样性。由于有各种不同的编程语言、软件包、操作系统和计算机硬件，软件可靠性工具不可能支持所有可能的组合。典型的例子就是，工具使用一种特殊的语法来开发，通常不同于实际编程用到的语法。因此，要使用形式化方法，往往要先将实际的软件转换为相应的工具能够识别的语法。

将软件系统自动转换为由某种固定形式表示的模型，有时可以通过一些编译技术来实现。但是，由于形式化方法的规约语言往往在结构的数量上比编程语言有更严格的限制，所以直接的转换通常不可能或者不实际。自动转换会导致大量无效的表述，使得某些目标技术或工具可以利用的优化方法无法顺利使用。因此，转换代码使其适用于形式化方法工具的限制，一般需要有经验的工程师的人工辅助。

对于一个实际系统的缩小版本，模型并非必需的。很多情况下，某个系统的模型在系统开发前或并行于开发过程中构造。这样做可以使得系统在初期就被检测，比如，在编写实际代码前先对设计进行测试。人们可以对初步模型进行可行性研究，或者检验非形式化规约的一致性。这个模型可以被具体化为实际代码。当系统的实质部分构建完成时，检验实际系统和初步模型的一致性也是很有效的。

一个模型能够真实地反映实际系统的属性是很重要的。建模的目的之一就是简化和缩减被检验的系统。为了达到这个目的，一个模型可能只保留所建模系统的某些属性。实际上，有时候为同一个系统建立不同的模型是很有效的，这些模型保留了系统的不同类型的属性。

为系统建模是一件很微妙的任务。有很多因素会导致错误，比如处理并行关系，或是使用正确的抽象层次。如果模型是错误的，那么通过验证或测试该模型以满足某个属性就无法在实际系统上实现。我们还需要更多的技术来更好地保证系统和模型间的转换能够保留被验证的属性。但需要说明的是，找到一种完全保证转换正确性的方法是不可能的。

我们需要详细区分一般形式化方法著作中的“模型”和本书中特别提到的“模型”。数学模

型是一个概念，是满足某些约束和观察的一类对象。从这个意义上说，我们讨论的是自动机模型（automata model）或者交错模型（interleaving model）。另一方面，一个特殊的系统模型是指某种数学对象，表达了对物理系统的抽象。从这个意义上说，我们讨论的是X11窗口系统模型。不幸的是，这两种“模型”含义的使用在形式化方法的术语中根深蒂固，而我们需要同时使用到它们。

4.1 顺序系统、并发系统及反应式系统

系统的某个模型应该能够反映系统的某些人们感兴趣的属性。为了有效地为软件系统建模，我们必须通过分析来表达系统的典型特性。我们首先描述建模和随后的形式化分析中将会涉及的不同类型的软件系统。

一个顺序系统通常可以用其输入-输出关系来描述，例如，连接可能的初始状态及可能的计算结果的条件。这种关系是可以被表述出来的，比如应用一阶逻辑来描述。因此，验证顺序程序或算法的正确性通常意味着用以下方式证明一个断言：如果程序以满足一个确定条件的状态开始，之后程序最终能结束，且最终状态的程序变量和开始执行时的相关值满足某种给定的关系。（第7章将会形式化地表达这种规约并且给出一些相关的证明技术。）

很多并发系统使用并行化来提高计算的效率。并行化可以将计算代价分配到不同的计算机上以达到更快的执行时间。相对于仅有输入-输出转换器，并发系统通常更加复杂。其中包括相互作用的顺序组件，我们称之为进程。输入-输出关系通常不足以说明这类的系统的行为。

多道程序设计（multiprogramming）是并发计算的一种模式。早期的计算机通常使用调度器在一个处理器上模拟并发。其目的是让程序更好地利用昂贵的（在当时）计算机时间资源。这种做法使得机器可以在其他任务或进程等待一般较慢的输入或输出操作完成时执行更多的计算。这种做法还使得多人同时在同一台机器上工作，而且每个人都感觉是在用一台自己独有的机器。多道程序设计系统包含一个调度器，调度器控制如何分割各进程的执行时间，将一个进程与另一个进程交换以优先执行。

如今虽然多道程序设计还十分普遍，但很多应用程序已开始使用由多个进程组成的分布式组件来合作完成一项任务。这样的多处理（multiprocessing）不但可以加快计算速度，还可以对系统进行分配，以一致的方式同时服务于不同地域的人。在这样的系统中，输入-输出关系退居次位。实际上，有些软件系统，例如操作系统和航空订票系统，被设计为仅在一些诸如断电等异常情况下才会结束。在多处理系统中，进程在不同的处理器上执行，相关的处理器的速度将影响不同进程间的交互。当一个有 n 个处理器的系统执行 $m>n$ 个进程时会产生多种组合的情况，这些情况需要多道程序设计和多处理技术来解决。

如果一个系统关注在组件间或是系统与环境间的交互，我们称这种系统为反应式系统（reactive system）[90]。反应式系统可以是顺序的，也可以是并发的。描述或验证并发系统或反应式系统包括很多方面，例如对请求的响应，或服务的可用性。

举一个航空订票系统的规约作为例子。我们可以确定，如果一个航班的座位没有被订满而一个顾客想要订票，那他最终会订到票。但是，如果系统仅剩一张票，而有两个顾客都要订票，这时系统必须阻止两个顾客订到同一个座位的情况发生。或者，系统可以防止订票数超过某个阈值。一般来说，顾客在订票的时候是位于世界各地的。这样的航空订票系统应该包括多个软件组件，其中一些组件是自治的，这些组件通过一些通信媒介来互相合作以保持系统总体的一致性。这样的系统，我们称之为分布式系统。

相对于简单的顺序系统，并发系统通常更加难以建模及分析。它们包含了不同的相互影响的组件。在任意给定的时刻，可能有不同的系统活动以及可选择的继续计算的方式。由于计算组件的速度及负载差异，不同的并发组件间存在着相互竞争。在每次独立的系统执行中，这样的竞

争可能产生不同的结果。也就是说，即使是同一个执行点，继续执行的方式也可以多种多样。举例来说，当两个进程都可以改变某个共享变量的值时，或是两个以上进程在通信时，这种情况都有可能发生。这种现象称为*不确定性*（nondeterminism）。这使得并发系统的建模变得复杂，从而导致规约和验证也更加困难。

使用一个适当的抽象来描述一个系统是很重要的，这样的抽象能保留必要的细节，同时忽略一些不重要的方面。很多用于软件建模的数学形式化方法忽略了实际执行时间的信息，比如状态间转换的时间。相应地，这些方法仅保留了状态间的顺序。之所以这么做，是因为实际执行时间可能相对不重要一些（只要实际执行时间是合理的，我们假设那是可以独立测试的）。而且，时间信息很难分析，而且可能会由于系统的某些部分被替换而变化。因此，抽象掉时间信息的验证更加容易，同时对硬件变化有更好的鲁棒性。然而，有些系统可能存在一些特殊的时间约束。近年来，人们设计了很多用来处理实际执行时间的形式化方法，这些方法旨在处理一些时间关键系统（参见［6］中的例子）。

以下是一些对软件系统建模需要考虑的主题：

并发性的表示 当由不同的并发进程执行某个转换时，我们应该如何建模？

- 某一时间仅允许一种转换（我们称之为*交错模型*）？
- 允许多个转换来改变状态（在*最大并发模型*中可行，这类模型主要用于硬件）？
- 使用事件间的偏序关系以反映事件发生的因果次序？

粒度 什么是合适的转换描述层次？是一个简单的赋值，一段待证程序代码中的判定谓词，一个在所有相关进程中执行的简单机器码指令，或是像执行代码的处理器的电压变化这样的物理变化？

执行模型 模型是否需要包括所有可能的完整执行的集合？或者说，由于不确定性的选择或是可能与环境交互，我们是否需要观测程序执行中所有点的所有可能分支？

全局或本地状态 在并发系统或分布式系统中，我们是否需要全局状态来表示整个系统的瞬时状态，或者只是使用本地状态来表示一个简单的并发处理器上某个变量的赋值？

4.2 状态

状态（state）是描述一个系统的核心概念。状态表示了程序在执行中的某个确定时刻的一些信息。状态可以是简单的描述程序的某个给定点的抽象实体，例如用 waiting_for_input_from_user 这个命题来表示程序的状态，这个变量在执行过程中的任意点上的值可以是 TRUE 或 FALSE。通常来说，一个状态可以用程序的每个变元的一种赋值来表示（从一阶逻辑的角度而言，参见 3.2 节）。有些时候，人们需要表述一些程序中没有直接提及的附加变量，例如，用来表示程序控制流当前位置的程序计数器，或是用来表示进程间消息传递的消息队列。

举例来说，一阶逻辑是一种可以表示程序状态的属性的规约形式，我们之前在第 3 章已经学习过了。一个使用程序变元作为自由变元的一阶公式可以表达一组满足该公式的程序状态集合。其中每个状态都是程序各变量的一种赋值。我们应该使用一种合适的一阶签名和结构来帮助判定程序域中的属性。

命题型的公式同样可以用于描述状态。这就要求这些命题根据一些状态命题来定义。举例来说，我们也许对变量 x 和 y 的单个值并不感兴趣，但是我们会关心当 x 的值大于 y 的值时，状态得以区分。于是，我们会定义一个命题 p，当 $x>y$ 时为 TRUE，否则为 FALSE。

*初始状态*和*终止状态*是程序状态集合中的两个重要状态。如前文所述，顺序程序的正确性标准往往只和这些状态集合相关。其他的中间状态和程序求值过程中的执行点相符合。正确性证明在程序过程产生的断言中使用这些中间状态，并依此建立程序变量在执行开始和结束时的

关系，或是验证程序的可终止性。

要表达并发系统、分布式系统或反应式系统的属性，我们需要能够表示程序状态间的动态变化的方法。我们将在第 5 章中学习可以表示这样的动态属性的形式化方法。

4.3　状态空间

对于顺序系统和并发系统，建模为*转换系统*（transition system）通常是很方便的。转换系统可以描述系统的行为，看上去就像每个时刻只有一个原子转换被执行一样。相似的模型也被用于描述硬件系统。

在 4.4 节中，我们将会形式化地描述转换系统，在转换系统中，系统可以处于有限或无限数量的状态中的某一个。在每个状态上，系统可以执行一系列原子转换中的一个。这一系列转换就是该状态可行的（enabled）转换，其他转换是不可行的（disabled）。在每个状态中选择一个可行的状态，将系统转换为一个新的状态。只要至少有一个可行状态，则该过程不断继续。这种过程，我们称之为计算的*交错模型*。

系统的*状态空间*通过一个图 $\langle S, \Delta, I\rangle$ 来表示，其中 S 是状态的集合，$\Delta \subseteq S \times S$ 是状态间的转换关系，$I \subseteq S$ 是初始状态。我们也把这样的图称为*自动机*（automaton）。这样命名从操作的角度强调了模型。我们可以把它看做一个简单的机器，这台机器可以识别程序的执行。自动机的一次执行是一个序列 $s_0 s_1 s_2 \cdots$，序列以状态 $s_0 \in I$ 开始并且根据关系 Δ（即对于任意 $i \geqslant 0$，$(s_i, s_{i+1}) \in \Delta$）继续。这样的一次执行必须达到最大化，也就是说，或者执行序列是无穷的，或者达到一个无后继的节点。需要注意的是，由于执行序列可能是无穷的，我们并不真正用一个自动机来识别所有执行。这仅仅是一个可以用来帮助分析系统的数学模型。我们会对自动机模型添加更多的结构并在第 5 章中详细讨论。

根据需求和关联（这会在之后的章节中提到），我们可以为自动机添加附加组件。举例来说，可以通过为 Δ 中的每条边添加标签来标识执行的转换。在有限状态系统中，可能用一个命题变量的集合来标记每个状态，每个变量表示某状态的属性。比如说，这些变量可以描述某个过程属于程序中的某个片段，某个过程已经终止，或者某个特殊的变量的值高于某个确定的阈值。

关于交错模型，我们将在后面给出形式化的说明，它把系统的执行看成是一个状态的序列，称为*交错序列*。在交错序列中，一个时间点只有一个转换得以执行。它的名字说明了，如果系统是由多进程组成的，那么各进程的转换通过交错形成一个单线程的执行过程。因此，每个交错序列在状态图中形成一条路径。由于其简单性，人们更喜欢使用交错模型，而不是诸如偏序模型（将在 4.13 节中讨论）之类的其他模型。很多数学工具，例如自动机和线性时序逻辑都能够支持交错模型，这简化了专门的形式化方法的开发。

当我们试图解释为什么我们用交错模型来描述计算，特别是并发计算时，我们常常会提出下面这个问题：

> 实际上，不同进程间的转换可能会在时间上重叠，我们应该如何在交错模型中表示这个特性？

要回答这个问题，我们考虑 α 和 β 两个执行时重叠的转换。假设 α 和 β 执行时都是始于某个给定的状态。通常并发执行转换的效果也是*可交换的*（commutative），也就是说，无论我们选择什么样的执行顺序，α 先于 β 或 β 先于 α，两者的执行结果会导致同样的状态。这样的情况下，交错模型会允许两种执行方式，一种是 α 先于 β，另一种是 β 先于 α。实际上，由于执行 α 和 β 的机器的相对速度，或是由于执行这些转换的多道程序的机器的调度方法，α 和 β 可以以两种方式中的任一种执行。它们也可以同时执行，但这种可能性在交错模型中没有明确地描述。

在图 4.1 中，转换 α 将变量 x 的值增加 1，转换 β 将变量 y 的值增加 1。两个转换从状态 $x=1$

且 $y=1$ 上同时执行。当两个状态以任意顺序交错执行时，结果都是状态 $x=2$ 且 $y=2$。

还有一种情况，转换 α 和 β 属于不同的进程，但是它们不是可交换的。举例来说，当两个转换改变的是两个进程的公共变量时，这种情况就会发生。考虑图 4.2 中的情况，α 将变量 x 的值增加 1，转换 β 将变量 x 的值加倍，两个转换都可以在状态 $x=1$ 开始执行。当 α 先于 β 执行时，我们得到一个状态 $x=4$。另一种情况下，当 β 先于 α 执行时，我们得到 $x=3$。

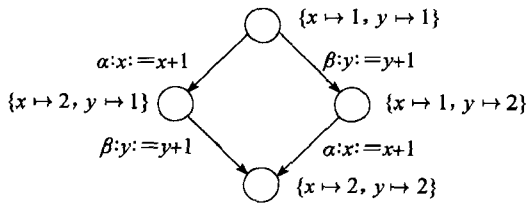


图 4.1 可交换转换的执行

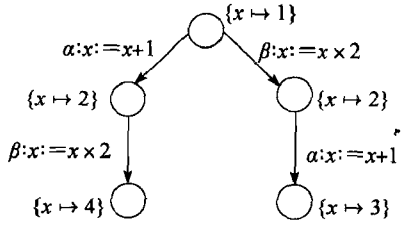


图 4.2 非可交换转换的执行

在这种情况下，α 和 β 的执行是不可重叠的，我们需要强制（通过硬件仲裁的方式）其按照某种顺序来执行。在交错模型中，如果不能够把并发的状态表述为一个接一个的执行的话，我们就不允许状态并发且相互影响。由于交互并不是软件建模所必需的，所以这通常不是一个大问题。有些系统，包括同步硬件，会允许以任何一种顺序执行的转换所获得的结果。这种情况下，我们可以选择另外的模型来表示这样的交互。举例来说，在硬件中，人们通常使用最大并发模型，这种模型允许多个转换在同一时间执行。有些时候，我们可以把所有这样的转换间的交互整合成一个大的转换，把所有可能的交互情况全部考虑在内。

4.4 转换系统

在本节中，我们将会定义*转换系统*。一个转换表示（顺序或并发）系统的一个原子操作（例如，初始化）。转换系统可以用来产生描述系统的状态空间。状态空间包含的信息包括系统的不同状态间如何相互联系，而转换系统包含了不同状态（转换）的产生者的信息。因此，一个转换系统在某种程度上暗示了其对应的状态空间。

从形式上来说，一个转换系统 $\langle S, T, \Theta\rangle$ 的定义如下：

- 一个一阶结构 $\mathcal{S}$，其中包括一个一阶签名 $\mathcal{G}$，一个域（或是多重域，即通常用于定义转换系统的结构的情况），关系及函数，还有一个解释函数。转换系统使用的条件和表达式通过签名 $\mathcal{G}$ 来表达，并且通过结构 $\mathcal{S}$ 来解释。其中签名 $\mathcal{G}$ 包含了一个有限变量集合 V。这个变量集合包括了程序代码明确描述的变量以及*程序计数器*（也叫做*位置计数器*）。每个并发进程都包含一个程序计数器，程序计数器通常隐含于程序代码中，它表示了一些程序的内部值，这些值指出了下一条指令或指令集的位置，进程可以选择这些指令或指令集继续程序的执行。在为系统建模的过程中，人们有时候需要为每个这样的程序计数器及其可理解的一组值（程序标签，goto 语句的目标，如果在原始代码中已经存在）的集合“发明”一些新的名字。这使得我们可以在程序变量的值相同而至少有一个进程处于不同的位置时区分不同的状态。
- 一个转换的有限集合 T，每个转换 $t\in T$ 以如下形式表示

$$p \rightarrow (v_1, v_2, \cdots, v_n) := (e_1, e_2, \cdots, e_n)$$

条件 p 是一个未测量的一阶公式，其变量属于 V。此外，v_1，v_2，…，$v_n\in V$，且 e_1，e_2，…，e_n 是结构 $\mathcal{G}$ 中的一阶表达式（项）。需要注意的是，每个程序只有一个转换集合，因此所有不同进程中的转换都会被放在一起。
- 一个初始状态 Θ，这是一个变量属于 V 的未测量的一阶公式。

我们也会把建模的程序中没有发生的结构$\mathcal{S}$函数和关系包括进来，同时在签名$\mathcal{G}$中添加合适的符号。之所以这么做是因为程序规约可能需要用到这些函数和关系这是因为程序的迭代性质会导致计算的数学结构并非用代码直接表示。举例来说，一个程序会使用迭代的加法来计算两个整数相乘。

直观上说，以 $p\rightarrow(v_1, v_2, \cdots, v_n) := (e_1, e_2, \cdots, e_n)$ 这样的形式表达的转换 t 可以在任何满足条件 p 的状态上执行。因此，条件 p 被称为转换 t 的可执行条件（enabledness condition），也可以表达为 en_t。当一个条件 p 在状态 s 上被满足，即当 $s\models^{S}p$ 时，我们说 t 在状态 s 上可执行（enabled）。在某个状态 s 上执行转换 t 的效果是基于状态 s 的变量的值计算各个项 $e_1, e_2, \cdots, e_n$。之后将获得的值分别赋给 $v_1, v_2, \cdots, v_n$，得到一个新的状态 s'。有些时候，我们通过在一个状态 s 上执行一个转换 t 来表示某种状态的变化，这样通过 $s'=t(s)$ 得到一个新的状态 s'。执行计算的顺序是很重要的。首先，所有的项都会根据当前的状态来计算，之后将计算的值赋给变量。因此，$(x, y) := (y, x)$ 的效果就是用 y 来代替 x。如果我们不先计算每个表达式再将其赋给变量，则赋值的结果就有可能不同。特别要说明的是，把 y 的值赋给 x 再把 x 的值赋给 y，其结果是把 y 的旧值赋给了 x 和 y。

初看起来，允许在原子转换中进行多个赋值似乎违背了原子性的想法，因为直观上，没有冲突地执行一些改变似乎很困难。但是，考虑到程序变量包含了程序计数器，对一个变量的一个典型的赋值也会涉及将程序计数器置为一个新值，来指向执行进程中的下一个可执行指令。

系统的一个执行是一个状态的序列。它开始于初始状态中的某一个，即某个满足初始条件 Θ 的状态，之后通过选择并执行系统的转换来实现一个状态到另一个状态的推进。一个执行可以到达系统终止或死锁的状态。终止和死锁的区别，仅仅在于当执行不能从一个状态继续时，该情况是计划中的（例如，当程序完成任务且没有下一步任务时）还是非计划中的（例如，在某些任务完成前程序终止）。

一个执行可能无法终止。这种情况可能是有意的，或是由于无法在有限时间内完成程序的任务。当然，实际系统从来不会无限地执行，因为它们可能被中断，或是最终被新的程序取代。尽管如此，我们通常对系统的执行不做限制，认为其可以永远继续，这样的系统抽象通常是十分有效的。我们不允许执行在至少有一个转换可以执行的状态上结束。因此，这些执行就是状态空间中的*最大路径*。

在定义执行时，我们需要指出真正代码的实际执行和其数学表示的重要区别。了解这种区别是十分重要的，在本书中，“执行”是*一种数学构造*，它和程序的一种模型相关，而与运行实际代码无关。

形式化地表述，一个程序的一个执行是一个无限的状态序列 $s_0, s_1, \cdots$，其中 $s_0\models^{S}\Theta$，即执行的第一个状态满足初始条件。对于任意 $i\geqslant 0$，需要满足以下两种情况之一：

1. 存在一个转换 $p\rightarrow(v_1, v_2, \cdots, v_n) := (e_1, e_2, \cdots, e_n)$ 在 s_i 上可执行，即 $s_i\models^{S}p$。此外，令 $l_1, l_2, \cdots, l_n$ 为项 $e_1, e_2, \cdots, e_n$ 的值。则 $s_{i+1}=s_i[l_1/v_1, l_2/v_2, \cdots, l_n/v_n]$。

2. 不存在可执行的转换，即其条件 p 在 s_i 上被满足的转换。则对于任意 $j>i$，我们可以得到 $s_j=s_i$。

这种情况下，当没有可执行的转换时，序列通过无限重复最后一个状态而扩展为无限序列。当然，这只是一种技术上的转换以保证不会同时出现有限和无限序列。当然，我们还可以在转换集合 T 中加入一个新的转换 t，当所有其他转换都不可执行的时候，该转换可执行，且不改变状态，这样也可以强制所有的序列变为无限序列。因此，如果 T 中的转换 $t_1, t_2, \cdots, t_n$ 的条件分别是 $p_1, p_2, \cdots, p_n$，则我们可以把这个 t 定义为 $\neg(p_1\vee p_2\vee\cdots\vee p_n)\rightarrow(v_1 := v_1)$，其中 $v_1\in V$ 是任意变量。

在程序的某个执行中出现的状态，称为可达（reachable）状态。并不是状态集 S 中的所有状态都是可达的。考虑这样一个例子，程序中的所有变量都被定义为（有界的）自然数，S 中的状态对应程序变量的所有可能的自然数赋值。举例来说，我们要求变量 $y1$ 的值总是大于变量 $y2$ 的值，这可能就是一个不可达状态。通常来说，定义状态空间时不考虑可达性因素，同时，根据程

序变量的值域来定义状态集 S，而不考虑由程序代码所带来的变量间的特殊联系，会使得定义过程变得简单一些。这也使得状态不会出现在所有可能的执行中。

我们也会使用上述执行定义的变体。举例来说，把执行定义为一个状态和转换（或者其标签）的交错序列，这是一种包含更多信息的定义版本。我们将在后面给出一些例子，在这些例子中，这种模型中增加的信息将会十分有用。另一种变体则完全忽略状态而关注转换，将执行定义为一个转换序列。在执行的各方面属性中，我们通常感兴趣的是状态间的发展变化。在第 8 章将要讨论的进程代数中，重点将是执行的转换。

上述关于执行的定义可以用一个调度器（scheduler）来描述，这个调度器可以为一个程序产生交错序列：

A Scheduler

Start from some initial state ι such that $\iota \models^S \Theta$. Set $s := \iota$

loop: if there is on enabled transition from s, goto *extend*.

Pick up a transition t that is currently enabled at s.

Apply the transition t to the current state s,

obtaining a new current state $s := t(s)$.

goto *loop*.

extend: Repeat state s forever.

这个调度器是非确定的，因为它必须首先在可能的多个初始状态中选取一个。此外，在每个状态上，需要选取一个可执行的转换。而且，可能有多于一种的选择。当且仅当没有更多的可执行转换时，调度器达到其最后指令，该指令在最后状态上无限重复。

以上调度方式有一个微妙的问题，即调度器可能对一个或多个并发进程存在不公平的现象，重复地忽略这些进程的可执行转换，而选择其他的进程。要解决这个问题，我们可以在调度器执行中加入一些称之为公平条件的约束。这个问题将在 4.12 节中讨论。

并发程序通常由多道进程组成，这些进程间通常会通过一些诸如共享变量、同步或异步消息传递之类的机制进行交互。可以单独地为每个进程建模，之后使用一些合适的操作符将它们合并。举例来说，我们可以为每个进程构造一个本地的状态空间，之后为它们定义一个全局的乘积。这个全局乘积中，每个状态和其本地状态的集合关联，每个进程关联一个（在通信系统中，消息缓冲中的值，即其包含的消息也会被传输）。乘积中的每个转换对应一个单进程中的转换，或是对应一些进程间的同步转换（比如握手通信）。

需要注意的是，我们并没有限制一个转换只能属于单个并发的进程。实际上，改变某些本地或全局变量的转换通常属于某个特殊的进程。但是，一些转换要求多道进程要一起同步。举例来说，一个同步通信（例如 Ada 的会合（rendezvous）机制）需要两个进程同步。

和多道程序设计模型很类似，交错模型中没有任何两个转换是同时执行的。在这个模型中，有一个单独的时间线，时间线和执行所有进程的调度过程相关。在分布式系统中，存在着真正意义上的同时运行。此外，大多数时候，并没有一个覆盖整个系统的全局状态。我们之后将会讨论偏序执行模型。对于分布式系统这是一个更加现实和直观的执行模型，因此经常称其为“真正的并发”。该模型允许转换在存在因果关系时按顺序执行，而在转换可以并发时不遵循固定的顺序。它还在不同的并发代理（进程）中引入了本地状态。原子转换可以在本地状态上执行，也可以在本地状态组合上执行。

转换系统是程序的隐式表示，就像程序的代码一样。为了分析程序，人们可能会想要获得其状态空间。一般来说，并不总是需要明确地构造一个程序的全部状态空间的表示。实际上，只有当处理某些有限状态系统时，才可能产生状态空间。用于有限状态系统的自动程序验证通常构造程序的状态空间，并且用图论中的算法来进行分析，具体方法将在 6.1 节中展示。但是，即使是这样，由于状态空间通常会很大，人们也总是在寻找一些方法以避免构造全部的状态空间。

4.5 转换的粒度

建模中最常见的错误之一和选择原子转换的粒度有关。这种选择是很微妙的。当转换的粒度太小时，例如在门电路一级描述转换，则这些描述包含的信息量超出了我们可以处理的范围。当允许处理更多表示每个状态的信息时，我们可能会区分出更多的状态，而这些状态的数量远远超出我们分析系统所需要的数量。这就导致了状态空间爆炸的问题，我们将在后面介绍。另一方面，如果选择的粒度过大，我们也许会漏掉一些进程间交互或是系统与环境交互的信息，从而导致结果无法覆盖所有可能的系统行为。

为了证明选择合适的粒度级别的困难性，让我们考虑这样一个例子，两个进程 P_1 和 P_2 需要在某个满足 $x=2$ 且 $y=3$ 的状态上执行以下指令：

$$P_1: x := x + y; \qquad P_2: y := y + x;$$

然后，让我们考虑一个多处理的实现，使得这两个进程在一个单顺序计算机上共享时间。变量 x 和 y 分别存储在寄存器 $r1$ 和 $r2$ 上，且使用的机器可以计算任意两个寄存器的和并且把结果存在第一个寄存器上。从汇编语言的角度上来看，我们可以获得如下代码：

$$P_1: \text{add } r1,r2; \qquad P_2: \text{add } r2,r1;$$

如果每个转换表示一条汇编代码指令，那么根据执行这两种加法的执行顺序，我们会获得两种可能的执行。第一种执行会先将 $r2$ 的值 3 加到 $r1$ 中，得到结果 5，之后把这个新的值加到 $r2$ 上，得到 $x=r1=5$ 和 $y=r2=8$。另一种执行则先将 $r1$ 的值 2 加到 $r2$ 中，得到结果 5，之后把这个新的值加到 $r1$ 上，得到 $x=r1=7$ 和 $y=r2=5$。在这个例子中，正是由于硬件不允许两个转换同时执行而导致了这样的情况。

假设现在变量 A 和 B 存在主存中而非寄存器中，其位置分别为 $m100$ 和 $m111$。汇编代码将会是如下形式：

P_1:	load $r1,m100$	P_2:	load $r2,m111$
	add $r1,m111$		add $r2,m100$
	store r1,m100		store $r2,m111$

如果每个转换对应一条汇编语言指令，则有 20 种执行方式。其中某些交错方式会得到和之前例子相同的最终结果。但是，这个例子多了一种可能的情况。以下的序列是以状态和转换交错的形式表示的。我们假设寄存器 $r1$ 和 $r2$ 的初始值为 0。

$$\{r1 \mapsto 0, r2 \mapsto 0, m100 \mapsto 2, m111 \mapsto 3\}$$
$$\text{load } r1, m100$$
$$\{r1 \mapsto 2, r2 \mapsto 0, m100 \mapsto 2, m111 \mapsto 3\}$$
$$\text{load } r2, m111$$
$$\{r1 \mapsto 2, r2 \mapsto 3, m100 \mapsto 2, m111 \mapsto 3\}$$
$$\text{add } r1, m111$$
$$\{r1 \mapsto 5, r2 \mapsto 3, m100 \mapsto 2, m111 \mapsto 3\}$$
$$\text{add } r2, m100$$
$$\{r1 \mapsto 5, r2 \mapsto 5, m100 \mapsto 2, m111 \mapsto 3\}$$
$$\text{store } r1, m100$$
$$\{r1 \mapsto 5, r2 \mapsto 5, m100 \mapsto 5, m111 \mapsto 3\}$$
$$\text{store } r2, m111$$
$$\{r1 \mapsto 5, r2 \mapsto 5, m100 \mapsto 5, m111 \mapsto 5\}$$

在这种情况下，最后 $x=m100=5$，$y=m111=5$。首先这两个变量的值在内部被读取，之后相加，最后再存回主存中。如果我们在系统的模型中选择和第一个例子一样的粒度，这个例子是和加法的高

级抽象一致的，那我们就可能忽略一种在实际汇编指令之间可能的交互情况。在用较大的粒度建模之后再验证代码，由于有一些额外行为无法用这个级别的粒度表示出来，所以这些额外的行为无法被发现，从而导致错误的出现。因此，即使错误发生了，验证程序也可能认为没有发现错误。

再举一个例子，考虑一个用更大粒度实现的系统，但其模型对应于上述较细的粒度级别。那么，模型中就会包含一些实际代码不允许的行为。这些额外的行为可能是错误的（在上面的例子中，规约不允许 x 和 y 的值在执行以后都为 5），并因此被形式化方法工具认为是反例。这些错误的警报称为误报（false negative）。

4.6 为程序建模的例子

为系统建模是使用形式化方法的一个重要部分。很多情况下，建模是人工完成的。由于不同的程序，特别是并发程序，使用不同的域和程序构造，所以并没有一个通用方案来为程序建模。我们将在本节中通过为一些类型的程序建模来证明这一点。

4.6.1 整数除法

第一个例子是一个顺序的程序，该程序的作用是计算 $x1$ 的值除以 $x2$ 的值的整数除法。除的结果在执行的最后存在变量 $y1$ 中，而余数存在 $y2$ 中。我们会给出这个程序的完整带标签版本。

$m1$：$y1 := 0$；

$m2$：$y2 := x1$；

$m3$：*while* $y2 \geq x2$ *do*

$m4$：$y1 := y1 + 1$；

$m5$：$y2 := y2 - x2$

$m6$：*end*

首先，我们需要选择合适的签名。程序包含了变量 $\{y1, y2, x1, x2, pc\}$，功能运算符“+”和“−”，以及关系运算符“≥”。程序还包含了两个域：自然数域，以及一个有限的标签域，即 $\{m1, m2, m3, m4, m5, m6\}$。类似地，我们可以用整数域来替代整数。我们还可以把标签编码为自然数 1…6。变量 $y1$，$y2$，$x1$ 和 $x2$ 可以被赋予自然数类型的值，而 pc 可以被赋予标签类型的值。功能运算符和关系运算符在自然数域的作用和其对应的数学符号相同。我们可以允许对变量 $x1$，$x2$，$y1$ 和 $y2$ 的自然数赋值，这是对以上程序的状态集合 S 建模的一种方法。此外，我们也允许从集合 $\{m1, m2, m3, m4, m5, m6\}$ 中选取一个值赋给变量 pc，用来表示程序计数器。满足初始状态条件 $\Theta : x2 > 0 \wedge pc \equiv m1$ 的状态即为初始状态。

可以看到，并非所有 S 中的状态都是可达的。仔细观察程序，我们可以看到一些变量间的联系。例如，当 $pc \equiv m4$ 的时候，我们可以得到 $y1 \times x2 + y2 \equiv x1$。这样的联系在之后的程序验证过程中将会有很大作用。但是，在建模过程中，人们可能会使用一个比实际可达状态集合更大的状态空间。转换系统包含了标记 $if(b, e_1, e_2)$，其中 b 是一阶公式，e_1 和 e_2 是项。如果 b 的值为 TRUE，则该标记返回值为 e_1；如果 b 的返回值为 FALSE，则该标记返回值为 e_2。

$$pc \equiv m1 \rightarrow (pc, y1) := (m2, 0)$$
$$pc \equiv m2 \rightarrow (pc, y2) := (m3, x1)$$
$$pc \equiv m3 \rightarrow pc := if(y2 \geq x2, m4, m6)$$
$$pc \equiv m4 \rightarrow (pc, y1) := (m5, y1 + 1)$$
$$pc \equiv m5 \rightarrow (pc, y2) := (m3, y2 - x2)$$

为程序的转换建模还有其他的选择。举例来说，可以用一对状态来替代第三个转换，每个状态控制程序转向不同的程序标签，如下所示：

$$pc = m3 \wedge y2 \geq x2 \rightarrow pc := m4$$
$$pc = m3 \wedge y2 < x2 \rightarrow pc := m6$$

根据代码可能会运行于其上的实际机器的配置，我们也可能对于可存储的值加以限制。实际上，对于本程序建模中用到的抽象，其中程序变量的值为任意自然数，有可能无法表示一个固定字大小的程序的某些实际执行：如果一个变量的值恰好超出了为每个变量分配的固定字大小所能表示的范围，则可能得到一个不同于以上转换集合约定的行为。这种情况下，程序可能返回一个错误消息并退出，或是产生某些无法预料的结果。进一步观察程序，我们可以发现，这种潜在的问题在以下这种特殊代码中绝对不会发生：一旦变量 $x1$ 和 $x2$ 的初始值和其分配的字大小相匹配，则其他变量 $y1$ 和 $y2$ 的值也会匹配（假设每个变量分配到的字大小相同）。

以下是一些可能的解决方法：

1. 忽略字大小的问题。假设用户永远不会使用过大的数字。由于使用这种模型的验证并不能覆盖使用过大数的情况，因此用户需要自己为使用过大数负责。

2. 为可能发生的值溢出建模。这样做可能要修改转换集合，使其能接受在真正的计算机上运行此代码产生的错误行为。举例来说，增加一个转换，使其可以在溢出时执行，之后模拟这种情况下计算机的异常处理行为。之后，验证一个抽象层次较低的模型，证明溢出情况不可能发生。

4.6.2 计算组合数

下一个程序（该程序对［92］中的程序做了轻微的改动）计算从 n 个不同的元素中取出 k 个不重复的数的所有可能组合的数量。该计算使用如下公式：

$$\binom{n}{k}=\frac{n\times(n-1)\times\cdots\times(n-k+1)}{1\times 2\times\cdots\times k}$$

该程序由两个并行进程组成。左边的进程在 $y1$ 的值处于 n 到 $n-k+1$ 的范围时，对分子进行累乘计算。右边的进程在 $y2$ 的值处于 1 到 k 的范围时，用累除计算分母的值。在执行的最后，计算结果储存于 $y3$ 中。

$$\begin{array}{l} l_1 : \mathit{if}\ y1=(n-k)\ \mathit{then}\ \mathit{halt} \\ l_2 : y3 := y3\times y1 \\ l_3 : y1 := y1-1 \\ l_4 : \mathit{goto}\ l_1 \end{array} \quad \Big\| \quad \begin{array}{l} r_1 : \mathit{if}\ y2=k\ \mathit{then}\ \mathit{halt} \\ r_2 : y2 := y2+1 \\ r_3 : \mathit{await}\quad y2<=n-y1 \\ r_4 : y3 := y3/y2 \\ r_5 : \mathit{goto}\ r_1 \end{array}$$

初始条件为

$$\Theta : y1\equiv n\ \wedge\ y2\equiv 0\ \wedge\ y3\equiv 1\ \wedge\ pc_l\equiv l_1\ \wedge\ pc_r\equiv r_1\ \wedge\ n>0\ \wedge\ k>0$$

在右边的进程中，标记为 r_3 的 *await* 语句使得标记为 r_4 的语句中的除法运算仅在当前计算分母所乘的数大于要除的数时才执行。这保证了除法能够产生整数的计算结果且不带余数。

我们使用的一阶结构包括函数符号“×”、“+”、“−”、“/”以及关系符号“≤”（和程序文本中的<=符号相对应）。表示相等的关系符号“≡”在一阶逻辑中已有介绍。这些函数和关系符号的作用和其在自然数域中对应的数学符号相同。

程序中大部分变量的值域都是自然数域，除了变量 pc_l 和 pc_r，这两个变量表示的是程序计数器。变量 pc_l 的值域为标签集 $\{l_1, \cdots, l_4, halt_l\}$，而变量 pc_r 的值域为 $\{r_1, \cdots, r_5, halt_r\}$。执行命令 *halt* 将终止进程的执行。集合中的特定标签 $halt_l$ 和 $halt_r$ 分别表示左边和右边的进程终止时程序计数器的值。该程序可以被翻译为如下的转换集合：

$$\begin{aligned} t_1 &: pc_l\equiv l_1 \rightarrow pc_l := \mathit{if}(y1\equiv(n-k), halt_l, l_2) \\ t_2 &: pc_l\equiv l_2 \rightarrow (pc_l, y3) := (l_3, y3\times y1) \\ t_3 &: pc_l\equiv l_3 \rightarrow (pc_l, y1) := (l_4, y1-1) \\ t_4 &: pc_l\equiv l_4 \rightarrow pc_l := l_1 \\ t_5 &: pc_r\equiv r_1 \rightarrow pc_r := \mathit{if}(y2\equiv k, halt_r, r_2) \end{aligned}$$

$$t_6 : pc_r \equiv r_2 \rightarrow (pc_r, y2) := (r_3, y2+1)$$
$$t_7 : pc_r \equiv r_3 \wedge y2 \leqslant n - y1 \rightarrow pc_r := r_4$$
$$t_8 : pc_r \equiv r_4 \rightarrow (pc_r, y3) := (r_5, y3/y2)$$
$$t_9 : pc_r \equiv r_5 \rightarrow pc_r := r_1$$

当然，这不是对该并发程序建模的唯一方法。在用转换集合为其建模时，我们假设了一种特别的粒度。当然，这些代码也可以被翻译成不同的转换集合，比如使用细致粒度。特别需要考虑的是转换 t_2 和 t_8。这两个转换都对 $y3$ 进行操作。和 4.5 节的例子类似，有可能程序被翻译为机器码，机器码首先将 $y3$ 的值读入内部寄存器，之后在该寄存器上进行计算（分别计算乘法或除法），然后把寄存器得到的新值存入 $y3$。这样的话，实际程序就存在着错误，而验证假设模型有不同的行为。因此，这个错误可能就无法被发现。

4.6.3 Eratosthenes 筛法

Eratosthenes 筛法（Sieve of Eratosthenes）是一种用于计算素数的算法。在其并行版本中，有一个最左进程用于生成整数，整数范围从 2 一直到某个上限 P。之后有 N 个中间进程，每个用于保存一个素数。第 i 个进程用于存储第 i 个素数。每个中间进程从其左边的进程接收数字。其第一个接收的数字是一个素数，这个素数被保留在接收进程中。之后的数将参照第一个数进行检验：如果从左边进程传过来的数能被第一个数整除，则这个数不可能是素数，因此这个数被丢弃；否则，该数字被传递给右边的进程。这种做法使得非素数被筛除。最右边的进程简单地接收左边进程传来的数。该进程保留第一个接收到的数（这个数是一个素数），而其他数被忽略。这种做法使得在最左边进程产生多于进程数量的素数时，系统可以允许数量的溢出。该算法用一个参数化的方式表述，其中的参数 $P>1$ 且 $N>0$。

命令 *ch* !*exp* 表示表达式 *exp* 已经被计算且通过队列 *ch* 发送。这是一种异步发送，其作用是将 *exp* 的值加入到队列末尾成为新的最后元素（如果队列未满）。当队列 *ch* 非空时，标记 *ch*?*var* 可以被执行。其作用是将第一个（即最早的一个）元素从队列 *ch* 中移除，并将其赋值给变量 *var*。

```
initially counter=2;
leftproc :: loop
      ch[1]!counter;
      counter := counter+1
until counter>P
||i=1,…,N
middleproc[i]:: ch[i]?myval;
while true do
     ch[i]?nextval
     if nextval mode myval≠0
         then ch[i+1]!nextval
end
||
rightproc :: ch[N+1]?biggest;
while true do
     ch[N+1]?next
end
```

在这里（以及之后的例子中）我们将不再赘述为程序建模的过程中如何选择标准的算术运算符及关系运算符及其表示法。不过，我们在这个程序中将为队列建模。我们用自然数列表来表示队列。为

了用转换系统给该模型建模，我们定义了一些函数，这些函数将帮助我们描述消息队列上的操作。

head：返回一个给定列表的头部，即第一个元素。（如果队列为空，则需要人为定义。）

tail：返回一个给定列表的尾部，即除去第一个元素的列表内容。（如果队列为空，则需要人为定义。）

append：该函数对一个列表和一个元素进行操作，用该元素加入列表中，使其成为最后一个元素。

列表中的元素用尖括号括住，如〈5，4，7〉。空列表表示为〈〉。

和集合类似，一般来说列表需要二阶逻辑表示。不过，我们在这里限制了列表的使用。举例来说，我们无法表示某些值属于一个列表（但我们能够表示某个值在列表的头部，更一般地说，对每个确定的 n，我们可以表示元素在列表的第 n 个位置）。

这里还需要说明几个建模时需要注意的问题。这个例子包含了一些参数化的并发进程，使用了 N 个中间进程和 $N+1$ 个消息队列。当通信队列显式参数化时，即 $ch[i]$，每个进程拥有自己的本地变量 *myval* 和 *nextval*。因此，在转换系统中，就有 N 个版本的这些变量分别和 $myval_i$ 及 $nextval_i$ 相关。为了将这些代码转变为转换系统，我们还需要为程序计数器变量以及程序位置常量起新的名字。这些名字也是参数化的，它们和中间进程的数量相关。需要注意的是，*haltleft* 的位置对应左边进程的终止点。

$$
\begin{aligned}
pc_l \equiv l1 &\rightarrow (pc_l, ch1) := (l2, append(ch[1], counter)) \\
pc_l \equiv l2 &\rightarrow (pc_l, counter) := (l3, counter + 1) \\
pc_l \equiv l3 &\rightarrow pc_l := if(counter > P, haltleft, l1) \\
pc_{m_i} \equiv m1_i &\rightarrow (pc_{m_i}, myval_i, ch[i]) := (m2_i, head(ch[i]), tail(ch[i])) \\
pc_{m_i} \equiv m2_i &\rightarrow (pc_{m_i}, nextval_i, ch[i]) := (m3_i, head(ch[i]), tail(ch[i])) \\
pc_{m_i} \equiv m3_i &\rightarrow pc_{m_i} := if(nextval \bmod myval \neq 0, m4_i, m2_i) \\
pc_{m_i} \equiv m4_i &\rightarrow (pc_{m_i}, ch_{i+1}) := (m2_i, append(ch_{i+1}, nextval_i)) \\
pc_r \equiv r1 &\rightarrow (pc_r, ch[N+1], biggest) := \\
&\qquad (r2, tail(ch[N+1]), head(ch[N+1])) \\
pc_r \equiv r2 &\rightarrow (ch[N+1], next) := (tail(ch[N+1]), head(ch[N+1]))
\end{aligned}
$$

初始条件 Θ 为：

$$
ch[1] \equiv \langle\rangle \wedge ch[2] \equiv \langle\rangle \wedge counter \equiv 2 \wedge pc_l \equiv l1 \wedge \bigwedge_{i=1,\cdots,N} pc_{m_i} \equiv m1_i \wedge pc_r \equiv r1
$$

考虑一个 Eratosthenes 筛法的例子，其中 $N=1$ 且 $P=3$。当该程序的参数特别是进程的数量增长时，该例子的状态空间增长得非常快。这个问题称为状态空间爆炸，这也是自动验证的主要挑战。

该例子的部分状态的赋值 $s1 \sim s10$ 列在如图 4.3 所示的表格中。我们用一个特别的标记⊥来表示未初始化的变量。这也是为系统建模的通常做法。这些状态中，每个状态里的变量 *biggest* 和

状态	pc_l	pc_m	pc_r	$ch[1]$	$ch[2]$	*counter*	*myval*	*nextval*
$s1$	$l1$	$m1$	$r1$	〈〉	〈〉	2	⊥	⊥
$s2$	$l2$	$m1$	$r1$	〈2〉	〈〉	2	⊥	⊥
$s3$	$l2$	$m2$	$r1$	〈〉	〈〉	2	2	⊥
$s4$	$l3$	$m1$	$r1$	〈2〉	〈〉	3	⊥	⊥
$s5$	$l3$	$m2$	$r1$	〈〉	〈〉	3	2	⊥
$s6$	$l1$	$m1$	$r1$	〈2〉	〈〉	3	⊥	⊥
$s7$	$l1$	$m2$	$r1$	〈〉	〈〉	3	2	⊥
$s8$	$l2$	$m2$	$r1$	〈3〉	〈〉	3	2	⊥
$s9$	$l2$	$m3$	$r1$	〈〉	〈〉	3	2	3
$s10$	$l3$	$m2$	$r1$	〈3〉	〈〉	4	2	⊥

图 4.3　Eratosthenes 筛法（$N=1$，$P=3$）的一些状态

next 的值都为⊥，因此没有在表格中明确列出。由于只有一个进程的类型为 $middle_i$，我们省略了表格中状态变量 pc_{m1}、$myval_1$ 以及 $nextval_1$ 的下角标记 i。

4.6.4 互斥

Dijkstra 是最先关注多进程系统的正确性问题的计算机科学家之一。在他的影响深远的论文[37]中，他阐述了如何通过互斥协议的临时尝试序列来合理地解决进程间的互斥问题。在这个问题中，两个（或更多的）进程竞争进入一个临界区（critical section），即它们访问某个互斥资源的代码段。这个访问过程必须是排他的。举例来说，临界区可以涉及打印。显然，我们不能让两个进程同时访问同一个打印机。

如果不需要访问临界区，则一个进程会进行任意的本地计算，这些计算位于非临界区中。之后，当要进入其临界区时，该进程将会加入一些互斥协议。此并发协议的作用是保证多个进程在准许其中一个进入临界区前无法访问各自的临界区。当进程获取了访问临界区的权限时，该进程可能需要再次加入（另一部分）的临界区协议，这一次的目的是为了允许其他进程进入其临界区。我们将在后面的章节中形式化地说明互斥问题的一些属性。用非形式化的语言描述，人们想要达到以下目的：

- *互斥性*（exclusiveness）：任意两个进程都不能同时进入其临界区。
- *活性*（liveness）：如果一个进程想要进入其临界区，则该进程能在有限时间内被准许进入。

以下是一个这种协议的临时尝试：

boolean c1,c2 *initially* 1;

*P*1 :: *m*1 : *while true do*	*P*2 :: *n*1 : *while true do*
*m*2 : (* *noncritical section* 1 *)	*n*2 : (* *noncritical section* 2 *)
*m*3 : *c*1 := 0;	*n*3 : *c*2 := 0;
*m*4 : *wait until* *c*2 = 1;	*n*4 : *wait until* *c*1 = 1;
*m*5 : (* *critical section* 1 *)	*n*5 : (* *critical section* 2 *)
*m*6 : *c*1 := 1	*n*6 : *c*2 := 1
end	*end*

我们假设临界区和非临界区都不会改变变量 *c*1 和 *c*2。我们可以用以下的转换系统来表示该算法：

$$\tau_1 : pc_1 \equiv m1 \rightarrow pc_1 := m2$$
$$\tau_2 : pc_1 \equiv m2 \rightarrow pc_1 := m3$$
$$\tau_3 : pc_1 \equiv m3 \rightarrow (pc_1, c1) := (m4, 0)$$
$$\tau_4 : pc_1 \equiv m4 \wedge c2 \equiv 1 \rightarrow pc_1 := m5$$
$$\tau_5 : pc_1 \equiv m5 \rightarrow pc_1 := m6$$
$$\tau_6 : pc_1 \equiv m6 \rightarrow (pc_1, c1) := (m1, 1)$$
$$\tau_7 : pc_2 \equiv n1 \rightarrow pc_2 := n2$$
$$\tau_8 : pc_2 \equiv n2 \rightarrow pc_2 := n3$$
$$\tau_9 : pc_2 \equiv n3 \rightarrow (pc_2, c2) := (n4, 0)$$
$$\tau_{10} : pc_2 \equiv n4 \wedge c1 \equiv 1 \rightarrow pc_2 := n5$$
$$\tau_{11} : pc_2 \equiv n5 \rightarrow pc_2 := n6$$

$$\tau_{12}: pc_2 \equiv n6 \rightarrow (pc_2, c2) := (n1, 1)$$

初始条件 Θ 为：$pc_1 \equiv m1 \wedge pc_2 \equiv n1 \wedge c1 \equiv 1 \wedge c2 \equiv 1$。我们可以看到，在转换 τ_2，τ_5，τ_8 和 τ_{11} 中，每个临界区和非临界区被建模为一个简单的转换。这是一个微妙的建模的结果，使用该方法需要仔细考虑。在模型中，这些临界区和非临界区总是会终止的。我们可能想要构造一种可能性，使得这些区域从不终止（理论上来说，至少持续到下次电源中断前）。这就需要将这些区域改变为循环模型，该循环使用一个非确定的选择作为出口。举例来说，我们可以添加以下转换：

$$\tau_2' = pc_1 \equiv m2 \rightarrow pc_1 := m2$$

$$\tau_8' = pc_2 \equiv n2 \rightarrow pc_2 := n2$$

这些转换在相同的程序计数器的位置上循环。实际上，我们通常需要该算法在某种假设下工作，该假设认为用 τ_2 和 τ_8 表示的非临界区能够永远循环，而用 τ_5 和 τ_{11} 表示的临界区最终会终止。

这就导致了另一个问题：如果一个非临界区可以永远执行，那么它是否会永远推迟其他进程的所有转换的执行？要避免这种调度结果在数学上的可能性，我们需要停止进一步的考虑，转而形式化地分析这种不公平的交错序列。这也是 4.12 节中定义公平性的动机。

练习 4.6.1 为互斥算法构建有限状态空间。

练习 4.6.2 图 4.4 给出了荷兰数学家 Dekker 对于互斥问题的解。将 Dekker 的算法翻译为转换系统。

```
                    boolean c1 initially 1;
                    boolean c2 initially 1;
                    integer(1..2) turn initially 1;

P1:: while true do                    P2:: while true do
  begin                                 begin
    noncritical section 1                 noncritical section 2
    c1:=0;                                c2:=0;
    while c2=0 do;                        while c1=0 do
      begin                                 begin
      if turn=2 then                        if turn=1 then
        begin                                 begin
          c1:=1;                                c2:=1;
          wait until turn=1;                    wait until turn=2;
          c1:=0;                                c2:=0
        end                                   end
      end;                                  end;
    critical section 1                    critical section 2
    c1:=1;                                c2:=1;
    turn:=2                               turn:=1
  end                                   end
```

图 4.4 Dekker 的互斥解

4.7 非确定性转换

在我们之前的定义中，使用转换系统为软件建模，每个状态上的可执行转换可以精确地获得一个直接的后继状态。因此，这种模型中的转换是确定的。这不表示建模的系统是确定的：有可能在相同的状态上执行不同的转换时有非确定性的选择。

但是，在某些情况下，为了方便人们也会考虑使用非确定性转换，即在某状态上执行该转换时，会有多于一个的可能后继状态。举例来说，考虑从用户处获取输入的情况。要表述这种情况，我们或许可以将每一个可能的输入值表示为一个独立的转换，或者为了方便，我们可能选择用一个单独的非确定性转换来表示，该转换集合了所有可能的输入。

要表示非确定性转换，我们需要对模型的定义做一些轻微的改变。以前，一个转换包含了一个可执行条件和一组赋值。现在，我们新增加一个变量集合 V'，该集合包括了 V 中的每个程序变量的变化版本。因此，如果 $V=\{x_1, x_2, \cdots, x_n\}$，则 $V'=\{x_1', x_2', \cdots, x_n'\}$。其中，$V'$ 中的变量表示程序中的同名变量在执行转换前的值，而 V' 表示相关程序变量在执行转换后的值。每个转换 t 现在表示为一个一阶公式 r_t，这个一阶公式说明了程序变量在执行转换前后的关系。因此，当我们可以对集合 V 中和状态 s 相关的变量赋值，且集合 V' 中的某些值满足转换条件 r_t 时，我们认为转换 t 在状态 s 上可执行。要用转换执行之前的变量值来表达转换 t 的可执行条件，可以用以下公式：

$$\exists x_1' \exists x_2' \cdots \exists x_n' r_t$$

需要注意的是，在对 V' 中的所有自由变量的存在性量化之后，可执行条件只包括了 V 中的自由变量。

考虑如下情况，将变量 $V=\{x, y\}$ 上的非确定性转换用如下公式表示：

$$x > y \wedge x' = x+1$$

这样我们可以得到 $V'=\{x', y'\}$。程序变量的域为整数域。我们可以将转换的可执行条件 $\exists x' \exists y'(x>y \wedge x'=x+1)$ 简化为 $x>y$。此转换的作用是增加变量 x 的值。因为转换未对 y' 做任何限制，所以 y' 可以为任意值。也就是说，我们可以从用户处获取任意整数值赋给 y'。如果这样做不符合程序的意图，而实际上 y 的值不发生改变，则我们必须将转换的公式表示为：

$$x > y \wedge x' = x+1 \wedge y' = y$$

这是一个确定性转换，我们可以用以前的标记来表示该转换：

$$x > y \rightarrow x := x+1$$

使用非确定性转换的形式，我们仅使用一个一阶公式，即可充分地表达所有可能的转换。特别是，假设 r_1，r_2，…，r_n 是（非确定）转换公式，则我们可以用一个公式 $\varphi = \bigvee_{i=1,n} r_i$ 来表达所有的公式。

4.8 将命题变量赋给状态

我们常常用形式化命题来为有限状态系统设置属性断言。为了将形式化命题用于系统建模，需要将规约中用到的命题和待建模系统的状态集合 S 关联起来。假设 AP 是一个有限的命题变量（有时候，我们称其为原子命题）集合。每个这样的变量表示某个待建模系统的固有属性，它可能属于或不属于系统的任意状态。每个状态可以被映射到其包含的命题子集中。

形式上，我们用 2^{AP} 来表示 AP 的所有子集（包括 AP 本身以及空集 $\varnothing$）。我们可以定义一个标签函数 $L_P: S \rightarrow 2^{AP}$，该函数将 S 中的每个状态映射到其包含的命题子集上。我们可以将这种带标签的转换系统表示为 $(S, \Delta, I, L_P, 2^{AP})$。

同样，每个状态也可以被看成是一种布尔赋值，将真值赋给 AP 中的命题。如果一个命题 p 属于 $L_P(s)$，则 s 将真值 TRUE 赋给 p；否则，赋值为 FALSE。这样，在前一种情况下，$s \models p$；在后一种情况下，$s \models \neg p$。这就给出了 L_P 的另一种定义，即一种将状态和命题映射到真值 TRUE 和 FALSE 上的函数，这种定义和前一种定义等价。我们可以这样重新定义 $L_P: (S \times AP) \rightarrow \{\text{TRUE}, \text{FALSE}\}$。

对于 L_P，还有另一种等价定义，我们可以把 AP 中的每个命题映射到包含该命题的状态子集上，重定义为 $L_P: AP \rightarrow 2^S$。这三种定义都是等价的，在与形式化相关的书籍中也都会使用。

状态的标签需要反映命题的一些概念意义。因此，如果一个命题 ye 关联的情况为交通信号

灯中的黄灯亮，则命题 *ye* 需要被精确地映射到反映该情况的状态中。命题到状态的不正确映射的后果，是检验结果可能变得无意义。

举例来说，让我们考虑 4.6 节中的筛法程序的状态。假设我们希望为程序设置的断言属性是左边进程的程序计数器是否是 *l2*，消息队列 *ch*[1] 是否非空。我们可以使用命题 *at_l2* 和 *nonempty_ch*1。我们需要一种方法来给状态做标记，使状态标签和其行为相关。在图 4.5 中，我们为状态 *s*1～*s*10 做了一种映射。读者可以将其和图 4.3 中的相关表格作对比。

状态	*at_l2*	*nonempty_ch*1
*s*1	FALSE	FALSE
*s*2	TRUE	TRUE
*s*3	TRUE	FALSE
*s*4	FALSE	TRUE
*s*5	FALSE	FALSE
*s*6	FALSE	TRUE
*s*7	FALSE	FALSE
*s*8	TRUE	TRUE
*s*9	TRUE	FALSE
*s*10	FALSE	TRUE

图 4.5　图 4.3 中状态的命题值

需要注意的是，多个状态可能拥有完全相同的命题标签。这说明了状态标签可能没有完全包含状态的相关信息。不过，它可能包含了足够用于验证的信息。因此，试着把映射到相同命题值的状态合并，有可能造成错误。

4.9　合并状态空间

我们可以从转换系统中获取状态空间。它可以表示被建模系统的全局状态，以及状态之间的转换。4.4 节中定义的转换系统包含了所有构成系统组件的转换。先构造不同系统组件的本地状态空间有时候更有利。举例来说，一个组件可以是一个并发进程，或者甚至是单个变量。之后，我们可以通过合并不同组件来获得全局状态变量。

考虑对转换系统的定义做一个扩展，用一些本地组件 T_1，…，T_n 来构造转换集合 T。每个转换集合中的 T_i 对应计算的一个组件，并表示诸如一个并发进程、一个共享变量，或是一个进程间通信队列。

此外，我们为每个状态集合 T_i 保留了一个转换名集合 Σ_i 和一个标签函数 $L_i: T_i \to \Sigma_i$，该函数是一个双射。这样的一个扩展转换系统并不是转换系统的一个转换 T 的简单部分。特别的，它是典型的属于不同集合的转换，如 T_i 和 T_j，但是被标记为同名的情况。

被标记为相同名字的转换总是在相同时间被一起执行。形式上，对于每个全局状态 s，我们可以用一个共享名 d 来执行所有 s 上的可执行转换（即对所有转换 t 有 $L_i(t)=d$）。执行这些转换的效果是同时执行转换上的赋值操作。因此，如果我们有两个转换 α 和 β，其中 $\alpha: p_\alpha \to (v_1^\alpha, \cdots, v_n^\alpha) := (e_1^\alpha, \cdots, e_n^\alpha)$，$\beta: p_\beta \to (v_1^\beta, \cdots, v_m^\beta) := (e_1^\beta, \cdots, e_m^\beta)$（且没有其他转换共享这两个名字），则我们可以在 $s \models p_\alpha \wedge p_\beta$ 的情况下从 s 上执行 α 和 β。这样，我们可以通过在 s 上使用一个多重赋值 $(v_1^\alpha, \cdots, v_n^\alpha, v_1^\beta, \cdots, v_m^\beta) := (e_1^\alpha, \cdots, e_n^\alpha, e_1^\beta, \cdots, e_m^\beta)$ 来获得一个新状态。

假设我们可以用一个和某个本地组件 T_i 相关的本地状态空间 $G_i = \langle S_i, \Sigma_i, \Delta_i, I_i \rangle$ 来表达系统的每个组件。我们将提供一种方法来通过合并不同组件的本地状态空间获取全局状态空间。我们假设不同组件的本地状态集合 S_i 是不相交的，但转换名集合 Σ_i 可能有非空交集。如果相同的转换名 α 同时出现在 Σ_i 和 Σ_j 中，则组件 G_i 和 G_j 在执行 α 的时候需要同步。执行这样一个**共享转换**（mutual transition），可以同时修改两个组件的本地状态。当一个组件表示一个进程，而另一个组件表示变量时，这样的合作可能和两个进程间的消息交换相关，或是和变量值的使用或更改相关。当然，一个共享转换可以被两个以上的组件共享，但是这种情况在为软件建模的过程中比较少见。如果一个转换的名称仅出现在一个单独的本地组件中，我们称之为**本地转换**。

我们需要一个操作符“$\circ$”来合并本地状态空间。要合并一对本地状态空间 $G_i \circ G_j$，我们可能希望其满足一定的条件，例如：

交换律 $G_i \circ G_j = G_j \circ G_i$

结合律 $(G_i \circ G_j) \circ G_k = G_i \circ (G_j \circ G_k)$

举例来说，这些条件意味着以下代码段的行为是相同的：

```
parbegin                parbegin                parbegin
  parbegin                parbegin                P2 :: ... end P2
    P1 :: ... end P1        P2 :: ... end P2      parbegin
    P2 :: ... end P2        P1 :: ... end P1        P1 :: ... end P1
  parend                  parend                    P3 :: ... end P3
  P3 :: ... end P3        P3 :: ... end P3        parend
parend                  parend                  parend
```

我们定义一个异步合并操作符“||”作为操作符“。”的一个可能实例，用来合并本地状态空间。这个异步合并与交错语义一致，表示不同组件的本地转换是交错的，即以某个任意顺序一一排列。

假设对于 $i=1$，2，有 $G_i=\langle S_i, \Sigma_i, \Delta_i, I_i\rangle$。我们定义 $G_1 \parallel G_2=\langle S, \Sigma, \Delta, I\rangle$ 如下：

- $S=S_1\times S_2$。每个合并的状态由一对状态组合而成，其中一个是 G_1 的本地状态，一个是 G_2 的本地状态。
- $\Sigma=\Sigma_1\cup\Sigma_2$。转换名包括属于 G_1 的转换名和属于 G_2 的转换名。（同时属于 G_1 和 G_2（即属于 $\Sigma_1\cap\Sigma_2$）的转换名也属于该并集。）
- 转换集合 Δ 由以下三个集合合并得到：

  a) $\{((s, r), \alpha, (s', r)) \mid (s, \alpha, s')\in\Delta_1 \wedge \alpha\in\Sigma_1 \setminus \Sigma_2 \wedge r\in S_2\}$。这种情况下，组件 G_1 发生变化，而组件 G_2 保持不变。

  b) $\{((s, r), \beta, (s, r')) \mid (r, \beta, r')\in\Delta_2 \wedge \beta\in\Sigma_2 \setminus \Sigma_1 \wedge s\in S_1\}$。这种情况下，组件 G_2 发生变化，而组件 G_1 保持不变。

  c) $\{((s, r), \gamma, (s', r')) \mid (s, \gamma, s')\in\Delta_1 \wedge (r, \gamma, r')\in\Delta_2\}$。这种情况下，两个组件同步，并一起执行转换。
- $I=I_1\times I_2$。合并的每个本地状态由一对本地初始状态组成，每个初始状态分别来自每个组件。

图 4.6 是两个本地状态空间的异步合并的例子。

需要注意的是，在硬件中，合并状态空间通常使用一种不同的方式：在执行的每一步，每个可执行转换的组件产生一个转换，全部在一步完成（不考虑转换名）。因此，前文中的异步合并操作符“||”并不是合并操作符“。”的唯一选择。

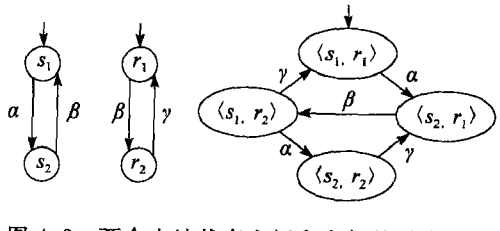


图 4.6 两个本地状态空间和它们的异步合并

从本地状态空间组合一个全局状态空间有很多优点。首先，这种做法为给系统建模提供了一个模块化的方法，使得人们可以分别注意每个组件。这么做提供了一种系统更加直觉化的表示，保持了系统的原始结构。这种做法还可以用来组合系统的规约，使其相对于一个巨大的全局规约更易理解。最后，一些软件可靠性方法可以利用这种模块化。例如，某些情况下，规定、测试和验证系统的本地组件更加容易。

4.10 线性视角

研究所有从初始状态开始的执行序列，是我们观察系统行为的一种方法。从系统的角度来说，我们总是希望每个交错操作能满足其期望的规约。一个规约可以表示一个允许交错序列集合 SP，这个序列集合可以由形式化规约给出。（我们将在第5章看到这样的形式化规约。）因此，如果一个程序 P 的执行集合为 EP，那么我们要求 $EP\subseteq SP$。如果存在程序的一个执行不属于

SP，那么这个规约是有冲突的。这一点可以通过如图 4.7 所示的 Venn 图证明。

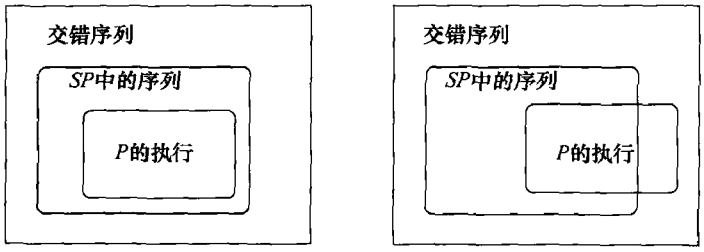


图 4.7　执行和规约之间的关系

根据线性视角，我们没有考虑不同执行间的关系。特别要说明的是，我们不关心是否两个有相同前缀的交错操作由于一些非确定性选择而在执行中的某些点产生差异。采用交错视角的原因是因为每次程序执行时，我们都希望其满足规约。如果有非确定性选择的影响，我们想要保证无论系统做何种选择，执行总是能满足规约。

4.11　分支视角

我们可以从另一个视角来研究系统的行为。选取一个初始状态$\iota \in I$，考虑从该状态开始展开的状态空间。这么做我们可以获得一棵（可能是无穷的）树，其节点是根据状态空间 G 中的相应状态标记的。树中的每个从ι开始的最大路径和程序的一个执行相关。如果树中的一个节点 s 有多个后继，则程序在 s 进行一个非确定的选择。当然，如果程序有多于一个的单初始状态，则我们可以看到多个分支树。

*分支语法*允许人们观测程序中需要进行选择的部分。计算树逻辑（Computational Tree Logic，CTL）[42] 是一种标准的形式化规约。它允许人们从树中的一个给定的状态描述，该描述可以是某些方式的所有执行行为，或是对于某给定方式存在一些执行行为。相对于交错视角，分支视角包含更多信息，因为人们可以通过忽略某些存在非确定选择的点来从分支视角获得交错视角。

分支视角的重要性在于，我们可以说明系统能够做选择。如果系统需要和另一个系统或是用户交互，则选择的可能性就显得非常重要。考虑一个自动售货机的例子，该售货机的每个物品价格为 60 美分。重要的不仅仅是每次当投入合适数量的钱时机器会给出一个零食。对顾客而言，他们感兴趣的是如果投入足够数量的钱，就可以在所有库存的零食中做出选择。

我们需要什么样的选择分支信息是很微妙的。尽管直觉上我们需要保证自动售货机允许用户选择，但是有些人可能会认为，我们并不需要使用系统分支化展开后获得的所有信息。因为对于每个点，只有当下的选择，或是从转换系统角度来说的可执行转换，才是人们关心的。因此，同样保持了每个节点的可执行转换信息的交错序列在这种情况下已经足够。更复杂的系统和环境间的接口可能包括了多级交互。有些情况下，我们需要更明确区分系统的行为和接口的行为 [7]。

关于线性视角和分支视角这两种范式，有很多的争论。以下的争论和上述自动售货机的例子类似，有些研究人员进行了区分，他们认为，*对于封闭系统*，即不和环境交互的系统，可以选择使用交错视角，而对于*开放系统*，即提供了各种交互行为的系统，应该使用分支视角。

线性视角和分支视角的区别可以进一步细化。并发理论领域（特别是进程代数领域）的研究人员区分了不同的分支模型。这些微妙的区别将在第 8 章中讨论。该章将证明选择正确的模型和合适的标准来对比不同的系统模型是很微妙的，取决于我们希望对所建模系统的行为观测的角度。

4.12 公平性

公平性（fairness）一般用于指在并发系统的交错执行上施加的语法限制。要讨论公平性，让我们考虑以下程序，该程序有两个并发进程 P_1 和 P_2，两个进程永久执行，且相互间从不交互（通过共享变量或消息交换）。表示程序执行的交错序列是一个由两个进程的状态组成的无限序列，其中每个状态是通过其前任状态执行 P_1 或 P_2 中的转换得到的。因为两个进程间没有交互，而且两个进程都永久执行，所以 P_1 和 P_2 总是有可执行的转换。每个交错序列中，在每个状态上，总是能够选择执行 P_1 上的转换或 P_2 上的转换。

我们可以在 4.4 节介绍的调度器中发现一个微妙的缺陷：在所有可能的执行序列中，有一些进程的可执行转换会被永久推迟。对于前文提到的程序，调度器可能在某些点仅选择执行 P_1 上的转换，或是仅执行 P_2 上的转换，尽管两个进程中的转换都是可执行的。

在多道程序设计的情况下，这相当于一个错误的调度器。实际的调度器会在进程抢占关键资源前限制其执行的时间片。在多处理情况下，缺少进程间的交互使得每个进程可以以各自的速度运行。因此，为了真实地表示程序的行为，每个执行需要为每个进程包括一个无穷数量的转换（当然，也可以是一个进程终止，而其他进程继续）。使用 4.4 节的调度器，这一点是无法保证的；该调度器还会产生永远忽略某进程的可执行转换的执行。

要对执行进行确切的分析，我们需要考虑实际使用的调度器，或是相关的处理器的速度。这显然是不实际的。我们不喜欢使用如此层次的细节来建模，就像我们不希望从晶体管或是门电路层次来解释程序的转换一样。此外，这样的分析会导致模型过于具体。将我们的测试或验证方法建立在如此具体的模型上会导致我们每次使用不同的操作系统或是硬件组件时都要重复进行分析。通常，我们分析系统是希望其可以运行在范围广泛的机器上，而不是一台特殊的机器。

公平性假设是用来排除对于我们建模的系统的体系结构不合理的无限执行。举例来说，人们可能希望通过公平性来阻止仅允许 P_1（或 P_2）执行有限次。形式化方法中有很多不同的公平性假设。以下将列出一些。对于以下的定义，我们需要重点区分被执行的状态和转换。因此，我们假设对执行的描述中，每对状态间执行的转换包含一个明确的指示（即名字）。

弱进程公平性。如果一个执行包含某状态 s，且总有至少一个属于进程 P_i 的转换可执行但在 s 之后没有 P_i 的转换被执行，则排除此执行。（P_i 的可执行转换可以在状态间改变。）

强进程公平性。如果在一个执行上进程 P_i 的转换可以被执行无限次，但仅被执行有限次，则排除此执行。

弱转换公平性。如果一个执行包含某状态 s，且总有一个转换可执行但从未在 s 后被执行，则排除此执行。

强转换公平性。如果在一个执行上有一个转换可以被执行无限次，但仅被执行有限次，则排除此执行。

公平性定义的多样性给我们提出了一个问题，即如何选择我们需要的定义。这实际上取决于我们想要建模的系统的特性。一种调度器可能符合一种公平性假设，而其他的调度器又可能需要不同的假设。需要注意的是，调度器和公平性假设相关，意味着所有调度器允许的执行是公平的，但反之不正确。实际上，对于很多公平性假设（包括前文列出的这些），人们可以在数学上证明：在没有提供特别的方法以产生真随机数的情况下，要构建一个允许完全公平执行的调度器是不可能的 [11, 47]。

要证明这一点以及前文提到的一些公平性条件，让我们考虑一个包含进程 P_1 和 P_2 的程序：

$$P_1 :: x := 1; \ \| \ P_2 :: \textit{while } y = 0 \textit{ do}$$
$$[\textit{no\_op}$$
$$[]$$
$$\textit{if } x = 1 \textit{ then } y := 1]$$
$$\textit{end}$$

我们假设这些进程有一个共享变量 x，其初始值为 0。进程 P_2 还有一个本地变量 y，其初始值也为 0。我们构造了新的名称 pc_l 和 pc_r 作为程序计数器。出现在方括号中的操作符“[]”表示一个非确定性选择。因此，进程 P_2 可以在一个 *no_op* 和一个 *if* 语句间选择。我们允许 pc_l 的值为 l_0 和 l_1，pc_r 的值为 r_0 和 r_1。我们用以下转换为这个程序建模：

$$t_0 : pc_l \equiv l_0 \longrightarrow (pc_l, x) := (l_1, 1)$$
$$t_1 : pc_r \equiv r_0 \wedge y \equiv 0 \longrightarrow pc_r := r_1$$
$$t_2 : pc_r \equiv r_1 \longrightarrow pc_r := r_0$$
$$t_3 : pc_r \equiv r_1 \wedge x \equiv 1 \longrightarrow (pc_r, y) := (r_0, 1)$$

初始条件为 $\Theta : x \equiv 0 \wedge y \equiv 0 \wedge pc_l \equiv l_0 \wedge pc_r \equiv r_0$。左边的进程 P_1 使用一个转换 t_0 进行建模，该转换将变量 x 的值置为 1。之后该进程的程序计数器置为 l_1，在该状态下没有可执行的转换。因此 P_1 在单独执行 t_0 后终止。进程 P_2 通过三个转换建模。转换 t_1 表示主循环的头部。该转换检验 y 是否等于 0。如果相等，执行进行到一个包含非确定性选择 t_2 和 t_3 的点。转换 t_2 是一个 *no_op* 语句。其作用是回到主循环的头部。转换 t_3 检验 x 是否为 1，若是则将 y 置为 1，之后回到主循环。进程 P_2 不断执行转换 t_1 和 t_2，直到 P_1 上的转换 t_0 将 x 的值变为 1。

弱转换公平性（以及类似的弱进程公平性）排除了一类执行，即进程 P_2 不断地执行 t_1 和 t_2，且永远不允许同时仅有 t_0 可执行的进程 P_1 继续执行。注意，在进程 P_1 上的转换 t_0 执行前，转换 t_3 不会变为可执行转换。现在，考虑 P_1 执行其仅有的转换的情况。执行后 x 置为 1。这种情况下，转换 t_1 和交替的转换对 t_2 和 t_3 变为可执行的。为了使进程 P_2 的循环终止，转换 t_3 必须被执行，将 y 置为 1。

弱转换公平性不保证终止性，因为在任何执行中，t_3 在任何状态下，永远都不会连续可执行。实际上，每次 t_1 可执行后，t_3 变得不可执行。强（或弱）进程公平性也不保证 t_3 会被执行。它允许在两个转换同时可执行的情况下，总是选择转换 t_2 而非 t_3。

在强转换公平性下，所有进程都能终止。转换 t_0 像之前一样能保证执行，终止进程 P_1。此外，在执行 t_0 之后，如果 t_3 未被执行，则进程 P_2 中的循环永不终止，t_3 则无限次地变为可执行。但强转换公平性要求在这种情况下 t_3 必须执行。因此，在任何情况下，t_3 都会执行。它将 y 置为 1 且程序计数器 pc_r 置为 r_0。之后，没有转换可执行，即 P_2 终止。

需要注意的是，我们通过否定的方法来使用公平性假设做推理。假设有一个 $P_1 \| P_2$ 的无限执行。这种情况下，我们使用公平性假设来排除 t_0 或 t_3 永远不被执行的情况。但是当 t_0 和 t_3 执行后，两个进程都终止，则排除了执行是无限的可能性。

我们现在将说明不存在一个调度器能够精确地产生对于以上程序满足强转换公平性的执行。如果有这样的调度器，我们可以在符合程序行为的分支视角的树中整理出一条公平的执行。该树在每个节点上有有限多个选择（最多为转换的数量）。如我们之前所见，这棵树是无穷的，因此有无限多的执行。但是，如之前所述，根据强转换公平性假设该树上的最大路径和一个有限执行相关。根据 König 定理的著名结论（参见 2.3 节）这是不允许的，König 定理断言，在一个每个节点有有限多个选择的无穷树中必然存在一个无限序列。相似的分析可以证明，不存在一个调度器满足前文提到的公平性假设。

以下提到的区分是非常重要的：上述证明只能说明人们不能够构造一个调度器来产生一个

给定程序的满足（强转换）公平性假设的所有执行。但是，我们可以确定能构造出一个仅产生唯一的（强）公平执行的调度器。这样的调度器因此可以产生公平执行的子集。

练习 4.12.1 描述一个调度算法，其作用是保证调度后的执行能满足弱进程公平性。解释你的算法为什么不能产生所有满足弱进程公平性假设的执行。

就像选择原子转换的粒度一样，公平性假设的选择对于正确地为并发程序建模也是很微妙的。在多道程序设计系统中，人们可能会试着验证一个调度器符合一些公平性假设。这也许是一件困难的任务（实际调度器可能包含很多时间细节，且调度器代码也许是对用户不可见的）。

我们没有一种通用的方法来决定使用哪种公平性假设。形式化方法的研究人员和从业者对于公平性假设的选择也是不同的。尽管某些人仅仅把公平性当做一个数学抽象的结果，但还是有很多人采用特殊的公平性假设且根据该假设规定并发系统的实际行为。

因为公平性假设是用于限制我们讨论的执行集合，所以使用正确的公平性限制对于获取可信的验证结果是很关键的。一方面，假设我们采用了一种对执行集合做出了过多限制的公平性假设。那么，验证此集合满足一个给定属性可能无法发现被移除执行中发生的错误。另一方面，假设我们使用了一种对执行集合限制不足的公平性假设。那么可能出现一种情况，即我们无法证实待建模系统满足某些属性，因为某些本来应该被更合适的公平性假设排除的执行仍然存在。

不幸的是，和其他建模问题一样，我们无法证明一个给定的公平性假设的确就是反映现实的正确选择。在即将在 4.13 节中讨论的偏序模型中，我们将给出一个更加直观的意义，使其因此对于一个特定的公平性假设有公正的结果。

人们可以在公平性假设中定义等级，即如果一个假设允许越多的执行，则该假设*越弱*。证明一个系统的正确性，最好是使用*最弱*的可能假设。更强的公平性假设允许的是更弱假设所执行的一个子集。因此，如果能证明满足一个更弱的公平性假设允许的所有执行拥有某个属性，则其子集，即更强的假设允许的执行同样能拥有该属性。当然，因为公平性仅限制允许执行的数量，因此如果能证明一个系统在没有公平性假设的条件下满足其规约，则在所有公平性条件下，系统仍然如此。

对于前文列出的公平性假设，弱进程公平性比强进程公平性弱（因此命名为*弱*和*强*），而弱转换公平性弱于强转换公平性。要理解这一点，我们可以考虑弱转换公平性和强转换公平性的例子（对于进程公平性，论证过程类似）。如果一个序列满足强转换公平性，那么对于每个转换，在某些点上，或者该转换永远不可执行，或者执行无限次。显然，对于这两种情况，弱转换公平性同样满足。

我们同样可以说明，在有限多个转换的假设下，强转换公平性强于强进程公平性。举任意一个满足强转换公平性的序列 ξ 为例。我们需要证明该序列同样满足强进程公平性。假设 P_i 是序列 ξ 的无限多个状态中的一个进程，且 P_i 中至少有一个转换是可执行的。因为每个进程有有限多的转换，所以 P_i 中至少有一个转换是会无限次可执行的。由于序列满足强转换公平性，因此在这个序列中会无限次地执行该转换。这意味着，在 ξ 中存在 P_i 中无限次发生的转换。对于任意 P_i，我们同样可以获得这样的推论。因此 ξ 满足强进程公平性。

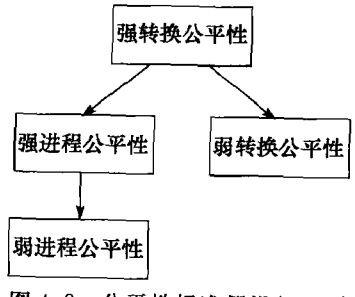


图 4.8 公平性标准假设的级别

我们可以证明，前文提出的四种公平性条件之间没有其他的关系。其等级如图 4.8 所示，其中从一个公平性条件指向另

一个的箭头说明前者强于（即能推出）后者。

4.13 偏序视角

相对于前文描述的交错执行，并发系统的偏序执行给出了一个对并发系统更直观的描述。实际上，如同我们之后将要看到的，程序的交错执行可以被理解为数学结构，它可以从程序的偏序执行中获得。尽管偏序执行模型更加直观，但交错序列的数学简单性使得人们也常常选择它来表示并发程序。虽然一些形式化方法技术可以直接推理偏序执行，不过交错序列在建模中还是更多地被使用。由于操作序列较为简单，因此使用交错模型来表示并发执行是很方便的。我们在这里描述这个模型的目的，是限制模型来更好地理解为并发软件建模。

对于交错模型，有一个批评的观点认为它无法区分非确定性选择和并发；正如我们之前在4.3节中所见，并发执行α和β的情况无法和先执行α然后执行β或先执行β再执行α的情况区分开来。由于执行原子转换的要求，使得交错模型对事件强加了全序。

此外，交错模型表示了在执行每个转换的前后整个系统的全局状态。实际上，一个系统很少经历这样的全局状态。由于不同的并发行为的发生，系统在同一时间并不只执行一个转换（除非是多道程序设计系统）。这通常不是一个大问题：系统的规约通常忽略强迫不同部分并发执行的限制。但是，正如我们将在本节的后面所要展示的，有些情况下，区分系统经过的实际状态和用交错模型建模的人工状态是十分重要的。

4.13.1 一个银行系统的例子

要说明交错模型的问题，让我们考虑银行中的两个支行。每个支行在工作日开始时有一百万美元的资金。在工作日中的某个点，一个客户向一个支行存入两百万美元。几乎是同时，另一个支行被抢劫，其拥有的一百万美元被盗走。大体上两个事件是同时发生的。要为此银行建模，我们让α表示存款的转换，β表示抢劫的转换。如果$x1$为第一家银行的资金数，且$x2$为第二家银行的资金数，则在α之前状态满足$x1=1\,000\,000$，且在α之后状态满足$x1=3\,000\,000$。类似地，在β之前状态满足$x2=1\,000\,000$，且在β发生之后满足$x2=0$。

如图4.9所示，我们用全局（状态）空间为当日的情况建模。此状态空间产生了两个交错序列，每个包含三个状态。

现在我们可以提问，那一天中，银行的实际资金是多少。银行的股东可能根据当前资金和初始资金的关系来决定买入或售出股票。假设他们使用一个网络代理程序来帮助决策，当资金多于两百万美元时买入，在资金达到或少于一百万美元时售出。当该程序通过两种不同的顺序了解到两种转换的发生时，它会做出不同的行为。如果存款消息先于抢劫消息通知到该程序，则程序会建议买入银行的股票。如果抢劫消息先于大笔存款的消息通知到该程序，则程序会建议售出股票，之后再建议买入股票。在这里，代理程序的行为可以视为判定投资好坏的规约。尽管两个交错序列最后关联到当日同时发生的一些特殊事件，规约还是会对银行的金融状况做出完全不同的评价。

这个假设的例子说明，处理交错序列和全局状态，有些时候会产生误导的效果。要更可靠地描述银行系统，我们需要注意每个支行各自的资金，从建模角度来说，就是其各自的本地状态。表示存款的转换作用在某个分支的本地状态上，产生一个新的本地状态，而表示抢劫的转换作用在另一个分支上。这种情况如图4.10所示。

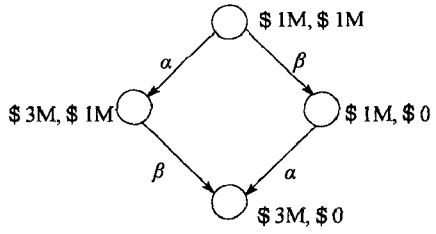


图 4.9 一个银行系统

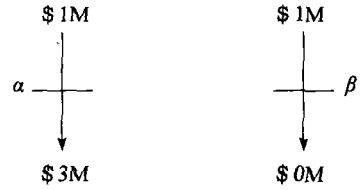


图 4.10 银行的偏序描述

一个*偏序执行*（E, $\prec$）包括一个有限或无限的事件集合 E。每个事件表示一个原子转换的发生。这些事件通过偏序关系“$\prec$”来排序。如果 $\alpha \prec \beta$ 则 α 在 β 可以开始前结束。实际上，偏序关系“$\prec$”意味着以下几点：

传递性 如果 $\alpha \prec \beta$ 且 $\beta \prec \gamma$，则 $\alpha \prec \gamma$。这一点直观地表示了如果事件 γ 在 β 之后发生，且 β 在 α 之后发生，则 γ 在 α 之后发生。

反自反性 $\alpha \prec \alpha$ 的情况从不发生。显然，一个事件不可能在自己结束后发生。

不对称性 如果 $\alpha \prec \beta$，则不能有 $\beta \prec \alpha$。因为如果 β 在 α 之后发生，则它不可能在 α 开始之前结束。

在交错模型中，每个事件和执行一个转换相关，事件间有一个全序（total order）。也就是，对于任何事件对 α 和 β，不是 α 在 β 之前发生，就是 β 在 α 之前发生。在偏序模型中，满足偏序条件 $\alpha \prec \beta$ 的事件在时间上是不能重叠的，即 α 必须在 β 可以开始前结束。此外，还可能存在相同的偏序执行，既没有 $\alpha \prec \beta$，也没有 $\beta \prec \alpha$。这意味着 α 和 β 在时间上可以重叠。但是，缺少顺序并不意味着两个事件必须重叠，尽管并发实现希望能利用两者执行的独立性。

让我们修改银行系统的例子，假设我们有一个包含三个事件 α，β 和 γ 的执行，其中 α 是财产存入事件，β 是抢劫事件，γ 是一个新的事件，它表示第一家支行的经理邀请存款者在当天共进晚餐。唯一的顺序对即 $\alpha \prec \gamma$，因此 α 和 γ 需要按照指定的顺序来执行。由于 α 发生（至结束）导致了 γ 发生，因此这个顺序表明了因果关系。事件 β 可以和 α 同时执行，和 γ 同时执行，或是和部分的 α 和部分的 γ 并发。因此，偏序模型描述了哪些事可以并发，但并不强制其同时发生。如果真的要说明 α 和 β 必须同时发生，我们可以定义一个单独的转换来表示这两个事件（或使用更丰富的执行模型）。

要将偏序执行模型和并发软件关联起来，我们观察到并发程序通常包含一系列组件，每个组件有各自的本地状态空间。举例来说，我们可能将以下每个对象设置为一个独立的组件，这些对象包括：

- 进程 P_i（包括仅能被 P_i 修改或使用的变量集合）
- 全局变量
- 消息队列

转换系统可以和在交错模型中一样定义。每个这样的转换包括一个依托于组件子集的条件。因此，在执行的偏序模型中，转换可以从一组本地状态上可执行，每个本地状态来自不同的组件，这样当这些状态一起达到时，我们获得转换的可执行条件。类似地，每个转换的执行可以修改一组不同组件中的状态。在前文银行系统的例子中，转换 α 和 β 仅根据每个组件的本地状态各自变为可执行的，用来表示银行的一个支行。但是，假设我们有转换 μ，其行为是每日结束时汇报一个支行的资金是否高于另一个支行，让我们考虑转换 μ 的可执行性。这个转换的执行取决于每个组件的本地状态。

考虑以下情况，即多个转换可执行，依赖于或改变至少一个共有的本地状态。这表示了一种

非确定性选择。举例来说，我们对银行系统的例子做一个扩展，假设有另一个人同一天在存入两百万美元的支行存入 50 美元。将这个事件标记为 δ。由于该支行只有一个出纳，两个存款操作不能并发执行，而由出纳决定哪个进程优先。两个存款操作都依赖于且改变一个支行的本地状态。

偏序模型能够区分由并发引起的非确定性和由一些代码中的选择语句决定的非确定性。在后一种情况中，有两个偏序执行。在银行系统的例子中，我们可以选择在 δ 前执行 α 或在 α 前执行 δ。这些选择对应不同的偏序执行。

4.13.2　线性化和全局状态

人们从每个偏序执行中获得相关的交错模型。这是通过将偏序关系补全为全序做到的（通常还有更多的方法可以做到），主要的操作是对无顺序的事件对加入顺序。每个通过这种方式从偏序中获得的全（即线性）序叫做对偏序的线性化。附加的顺序必须满足两个条件：

- 不会形成事件的顺序环
- 每个事件之前只有有限数量的其他事件。

因此，一个偏序执行可以关联到多个交错序列。

在执行的偏序模型中，我们可以通过从每个不同的组件中收集一个本地状态来获得一个全局状态。但是，并不是每个收集的结果都能形成一个全局状态。每个全局状态 s 将偏序执行中的事件分成两部分：发生在 s 之前的（根据事件间的偏序关系），标记为 $before(s)$；那些之后发生的，标记为 $after(s)$。

仅当集合 $after(s)$ 是有限且左封闭时，这样的分割才有意义。给定一个偏序执行 $(E, \prec)$，一个左封闭集合 $C \subseteq E$ 必须满足：

如果一个事件 α 属于 C 且对事件 $\beta \in E$ 有 $\beta \prec \alpha$，则 β 同样属于 C。

我们同样通过 $before(s)=C$ 来构造赋值 a 并附加到全局状态 s 上。我们假设偏序执行 $(E, \prec)$ 中的每个事件和系统的一个或多个本地组件（进程）相关，其中每个本地组件有一个初始状态。此外，对于每个事件，有一个本地状态表示相关组件中的变量在事件发生后的值。全局赋值 a 和 C 中的最大事件后的本地赋值相符。（对于事件 $\alpha \in C$，如果没有其他事件 $\beta \in C$ 满足 $\alpha \prec \beta$，则称 α 为最大事件。）如果对于某个组件（进程）C 不包含任何事件，那么对于这个组件的变量，全局赋值 a 和初始值一致。

考虑银行系统的初始版本和图 4.10。其中有两个无序事件 α 和 β。因此，我们可以得到 $E=(\{\alpha, \beta\}, \varnothing)$。第一个支行的本地初始状态为 $x1=1\,000\,000$。另一个支行的本地初始状态为 $x2=1\,000\,000$。在执行 α 之后，本地状态为 $x1=3\,000\,000$。在执行 β 后，本地状态为 $x2=0$。这里有 4 个左封闭子集：

- $\varnothing$（事件的空集）。因为空集中没有事件，两个组件（银行支行）处于其初始状态。全局赋值和本地初始状态一致，即 $x1=1\,000\,000$ 且 $x2=1\,000\,000$。
- $\{\alpha\}$。仅有一个支行参与。对于此状态的全局赋值有 $x1 \mapsto 3\,000\,000$，这和本地状态执行 α 后相符。因为另一个支行仍处于其初始状态，所以全局赋值同样有 $x2 \mapsto 1\,000\,000$，这和该支行的本地初始状态一致。
- $\{\beta\}$。仅有一个支行参与。对于此状态的全局赋值有 $x2 \mapsto 0$，这和本地状态执行 β 后相符。因为另一个支行仍处于其初始状态，所以全局赋值同样有 $x1 \mapsto 1\,000\,000$，这和该支行的本地初始状态一致。
- $\{\alpha, \beta\}$。两个支行都参与。由于 α 和 β 是无序的，所以两者都是最大的。全局赋值同 α 执行后的本地状态即 $x1 \mapsto 3\,000\,000$ 和 β 执行后的本地状态即 $x2 \mapsto 0$ 保持一致。

对于偏序模型的全局状态的定义允许我们将全局状态的概念关联回之前在交错语义上的定义。但是，需要注意偏序执行中的全局状态并不是一般意义上的线性顺序。实际上，从偏序执行获得的全局状态间的顺序，和事件集合的包含关系相符。也就是说，如果存在关系 $before(s_1) \subset before(s_2)$，则全局状态 s_2 在全局状态 s_1 之后发生。在前文的例子中，和 $\{\alpha\}$ 相对应的全局状态发生在和 $\varnothing$ 对应的全局状态之后，并且发生在和 $\{\alpha, \beta\}$ 对应的全局状态之前。但是，和 $\{\alpha\}$ 及 $\{\beta\}$ 相符的全局状态是无序的，且它们对应的赋值不能出现在同一个线性化中。

4.13.3 一个简单的例子

图 4.11 是一个简单程序的例子。该程序有两个进程 $P1$ 和 $P2$。进程 $P1$ 开始时有 $x=0$，它不断递增 x 的值，并且通过一个同步通信序列将新的值传给 $P2$。进程 $P2$ 开始时 $y=0$。该进程每次接受一个值，传给其变量 z，之后加到 y 上。为了使这个例子简单一些，我们忽略了一些实际因素，例如数据字长度。因此，我们假设两个进程永久运行。由于通信是同步的，因此有一个单一转换来表示通信，有效地将 x 的值赋给 z。这个转换被两个进程共享。因此，这个程序有三个转换：

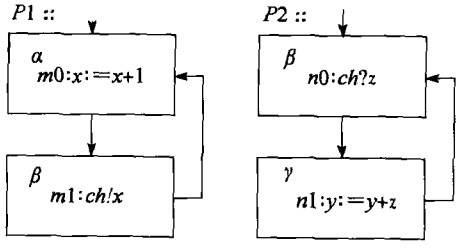


图 4.11 通过消息传递进行进程交互

$$\alpha: pc1 \equiv m0 \rightarrow (pc1, x) := (m1, x+1)$$

$$\beta: pc1 \equiv m1 \wedge pc2 \equiv n0 \rightarrow (pc1, pc2, z) := (m0, n1, x)$$

$$\gamma: pc2 \equiv n1 \rightarrow (pc2, y) := (n0, y+z)$$

图 4.12 描述了程序的一个偏序执行。它展示了状态发生间的顺序和每个转换前后的本地状态。因此，图中的箭头不直接表示事件间的偏序关系。相反，偏序由事件之间的路径来表示。举例来说，顶端事件 α 的发生先于 β 的发生，而 β 又在另一个 α 之前发生。因此，之前的 α 的发生同样在之后的 α 的发生之前。有些 α 和 γ 的发生是无序的，这和它们可以并发执行的可能性相符。但是，有些 α 的发生排序在一些之后的 γ 发生之前，反之亦然。

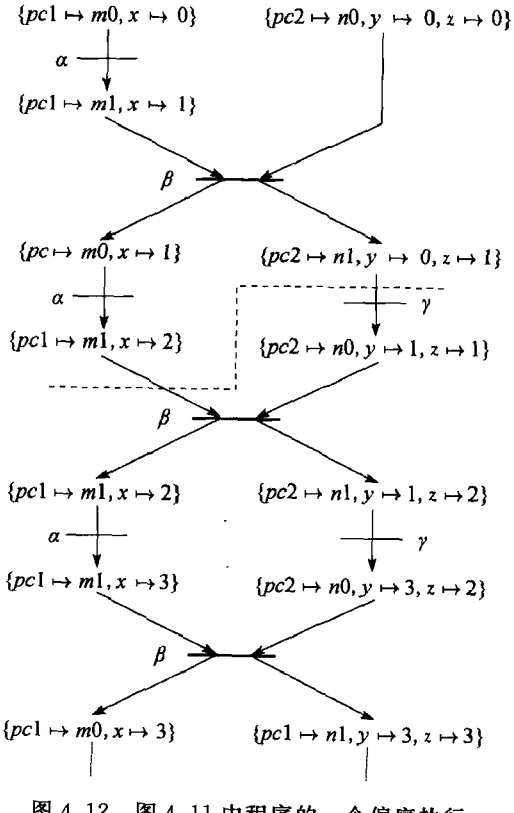


图 4.12 图 4.11 中程序的一个偏序执行

因为没有非确定选择，所以对于这个程序实际上只有一种偏序执行。图 4.13 给出了无限的偏序执行的补全的全序中的两种。图 4.12 中的虚线对应一个全局状态。这个状态包含了该虚线之前出现的所有本地状态。它和执行中的一部分相关，这部分包括两次事件 α 的发生和事件 β 的单次发生。相关的全局赋值可以通过获取虚线下面的最大本地状态的赋值来形成，即

$$\{pc1 \mapsto m1, x \mapsto 2, pc2 \mapsto n1, y \mapsto 0, z \mapsto 1\}$$

除了以上事件之外，一个之后的全局状态也包括了事件 γ 的第一次发生，该事件符合全局赋值

$$\{pc1 \mapsto m1, x \mapsto 2, pc2 \mapsto n0, y \mapsto 1, z \mapsto 1\}$$

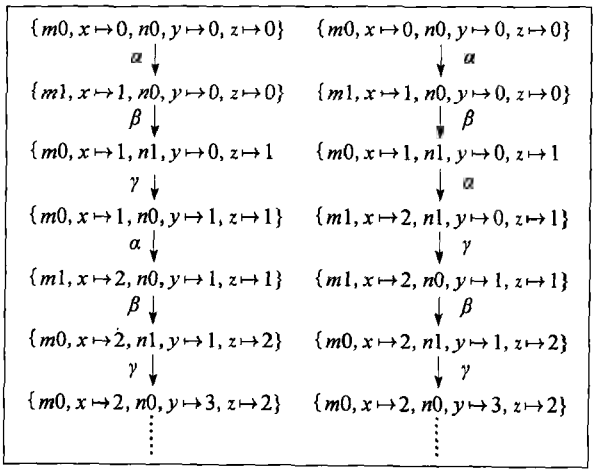


图 4.13 两个交错序列

4.13.4 偏序模型的应用

通过偏序执行和交错序列的相关性，我们可以将每个交错序列视为一个数学结构。这使得我们可以用一些数学工具来处理序列。有趣的是，我们发现，很多用来推理交错序列的形式化方法，包括时序逻辑，可以区分从相同偏序执行中构建出来的交错执行，同样，前文银行系统的例子中描述的代理程序也能够做到。

有一个有趣的关于偏序和交错的关系的应用，即基于偏序的验证 [76]。它发现了一个事实，即规约通常无法区分符合同一个偏序执行的线性化（交错操作），因此这样的区分必须人工操作。这样的验证方法，通过为每个偏序执行选择方便的代表来推理程序。因此，我们需要推理更少的可能性且验证通常变得更简单。与此相似，偏序约减 [55，113，142] 尝试在算法上利用这种现象来减少模型检验算法中发生的状态爆炸问题（参见第 6 章）。

和公平性假设对交错序列的限制相同，人们可能对偏序执行加以限制。最常用的限制叫做极大性（maximality）。它要求如果一组本地状态中没有一个立即跟随事件，则一个偏序执行不能激活这组本地状态共同的转换。

有趣的是，我们注意到表面上极大性限制可以被转换成相关的由偏序执行产生的交错序列上的公平性假设。定义一个（对称自反）的依赖关系 D，该关系作用于转换上，$\alpha D \beta$ 表示任何 α 和 β 不能并发执行。也就是说，不能有这些转换的无序（关于“$\prec$”）发生。这样我们就可以证明（参见 [82，115]）极大性等价于如下假设：

> 不存在一个无限的执行，对于某些状态 s，永远有一个转换 α 可执行，且永久保持可执行，但 α 和任何依赖 α（根据关系 D）都不会在 s 后执行。

此假设和弱进程公平性非常相似。这里我们再次重申，偏序和交错执行的相关性可以帮助我们解释建模问题。

4.14 形式化建模

很多软件可靠性工具中用于表示软件系统的形式化只是简单的程序设计语言。用户在使用工具前，必须编写系统的抽象描述。其他形式化建模是基于一些有限自动机的扩展形式。某些情况下，自动将系统的初始表达转换为形式化模型是直接可行的。但是，由于形式化方法的相关复杂性和可计算性限制，这种特性通常受到严格限制并且不是广泛可行的。因此，存在一个惯例，

即软件可靠性工具的用户以人工方式对待检验系统进行抽象，从而获得通过所用工具的特定建模形式化而给出的描述。

设计形式化建模是为了达到以下目的：

- 在形式化建模和某内部表达间存在一种简单、有效的转换。对于软件可靠性工具而言，通过优化，我们可以使用其内部表达来获得更佳的效率。
- 形式化将用户限制到一个形式化方法可以应用的范围。例如，它可以限制模型包含有限的状态，或使用一个特殊的签名。
- 形式化丰富且易于使用。为系统建模的过程通常由人工完成。这就成为使用形式化的瓶颈，且有很大可能造成建模错误。丰富的形式化可以简化建模过程且使得建模错误最小化。
- 形式化建模需要有一个清晰且形式化定义的语义。软件可靠性工具关注的是系统的正确性。因此，工具的开发人员需要努力使得工具不会引入含糊或是误导性的结构，因为这些结构会导致系统建模错误。

某些标准建模形式化的目的，是为了使得工具开发人员无需设计新的程序设计语言，同时获得统一且受到广为接受的语义。在描述通信协议的例子中尤其如此。这样的形式化通常被称为“形式化描述技术”（FDT）。ISO（国际标准化组织）定义了一些著名的标准形式化的例子。例如 ESTELLE[23]、LOTOS[17] 以及 SDL[62]。但需要注意的是，标准形式化的开发通常独立于特殊的软件可靠性工具。

进程代数也被用于为系统建模。有一种这样的形式化称为 CCS[100]，它是 LOTOS 的原型。CCS 和 LOTOS 将在第 8 章中讲述。另一种广受欢迎的形式化叫做 PROMELA[65]。这种程序设计语言在模型检验工具 SPIN 中使用。它允许大量的程序设计结构和不同类型的并发交互，包括同步和异步消息传递以及共享变量。对于 PROMELA 的关注和使之成为一种丰富的建模语言的努力，使得 SPIN 成为一个非常受欢迎的验证工具。

有时候，将一种工具使用的形式化建模语言转换为另一种工具使用的另一种形式化建模语言是可行的。这样的转换使得我们可以不用重复建模过程就能使用不同的形式化方法和软件可靠性工具。这是有可能的，因为许多工具关于计算模型使用了相同的假设，特别是具有有限多个状态。但是，需要注意的是，由于形式化方法通常基于计算难题，因此很多工具使用了各种启发式方法来根据它们使用的特殊形式化进行优化，使其工作效率提高。举例来说，一个工具可能包含一种启发式方法，该方法缩减了检验一种特定的通信模式所需要的状态数量。这样的模式之后可以在模型的编译过程中被识别。将一种形式化转换为另一种会忽略这样的启发式方法，且因此产生一种目标工具表现较差的系统描述。

另一种为系统建模的方法，是用*图形接口来获取系统的描述*。这种建模方法将系统的可视化描述放在一起，通常组成一个有限图。这种描述被转换为一些内部表达，对于用户这种操作是透明的。这样的描述特别适合于有限状态系统。无限状态系统同样可以通过类似的方法描述，这种方法叫做*扩展有限状态系统*：和直接表示系统的状态不同，人们可以表示有限数量的参数化状态，这些状态只包含部分描述。附加的参数，例如程序变量或消息队列，并没有可视化描述。这些可视化的成果近来变得十分重要，也成为软件可靠性工具变得越来越受欢迎的原因之一。

使用图和其他图表可视化的规约为我们在设计系统时提供了一个更直观的理解。举例来说，UML 方法学 [46] 以及相似的 ROOM 方法学 [127] 包含了一组不同的图，这些图表可以表示所设计的系统的不同方面。有一类图用来表示系统的体系结构，包括不同的进程和它们之间交互的方法。另一类图用于描述系统中每个进程的行为。这一般是通过使用一些类型的状态空间描述完成的。系统的不同行为通常通过其他形式化来描述，例如消息序列图，它描述不同场景的

消息交换。在第 11 章中，我们将进一步说明可视化描述技术。

4.15　一个项目的建模

以下通信协议的目标是在有损通信队列上传输一个消息。根据 OSI（Open Systems Interconnection，开放系统互联）通信模型，这个协议是数据链路层的一部分。发送者进程通过一个不可靠通信链路发送一个消息（使用命令 *SendEnv*）。所处环境可能最终将消息转发到接收者进程，或者消息会丢失。如果消息没有丢失，接收者进程通过 *EnvReceive* 命令接收消息。由于存在消息丢失问题，发送者进程通过一个比特位 *NextMessage* 来发送消息数量的指示。这个变量使用一个模块来计算被传送的消息数。从发送者发送到接收者的消息包括了消息的内容和 *NextMessage* 计数器的值。

由于没有标记显示消息丢失，发送者使用一个计时器，每次一个消息发送后开始计时，每当消息被接收时重置。接收者通过 *AckReceive* 命令等待消息到达的确认或是通过 *TimerUp* 事件等待计时器到时。后者触发发送者尝试重新发送。当然，如果接收者或者网络环境过慢，计时器会提早到时。在这里，我们需要对该算法进行形式化验证（或者相反的，证伪）。

等待一个事件通过使用结构 *wait-event* […] 在下面给出。如果有多个事件可以发生，它们会在结构中通过“[]”被分开。一个触发事件，即一个接收事件或一个超时事件，之后跟着一个“->”且这个指令必须在事件发生之后被执行。

```
Sender :: bool NextSend, AckVal = 0;
    while true do
     prepare NextMessage;
     SendEnv NextSend, NextMessage;
     StartTimer;
     wait event [
       AckReceive AckVal ->
         CancelTimer;
         if AckVal = NextSend then
            NextSend := 1 - NextSend
       []
       TimerUp -> no_op
    end
||
Receiver :: bool MsgNum, FrameExpected = 0;
    while true do
     wait-event {
       EnvReceive MsgNum, MsgContent ->
        if MsgNum = FrameExpected then
           FrameExpected := 1 - FrameExpected;
        ReceiverAck! 1 - FrameExpected}
    end
```

更多关于此协议的细节和类似的协议可以在 Tanenbaum 的关于计算机网络的书 [138] 中找到。

在对此协议建模前，以下几点需要考虑：

- 计时器可以被认为是一个单独的进程，该进程包括三个转换：设置、重置和到时。这些转换和发送者进程共享。
- 不需要强加一个实际时间约束来引起计时器到时。我们可以简单地通过一个包括到时和被发送者取消的非确定性选择来为其建模。
- 两条通信链路：从发送者到接收者的用于发送消息的链路，以及从接收者到发送者的用于接收消息的链路，可以通过与相关进程的转换共享，用两个额外进程建模。
- 实际消息在这里是无关的。一种可能是使用一个非确定性转换，对于和准备 *NextMessage* 相符的转换中赋给 *NextMessage* 的值，该非确定性转换不做限制。另一种可能是忽略和消息内容相关的部分。抽象的过程将在 10.1 节中进一步讨论，而且一般需要一些形式化解释（尽管实际上，通常是非形式化的解释）。

4.16 扩展阅读

以下书籍给出了系统建模的全面讨论：

Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1991.

一本全面关注建模（和验证）公平性问题的书是：

N. Francez, *Fairness*, Springer-Verlag, 1986.

以下这本书综述了开发程序设计语言和经典并发算法的例子，还进一步给出了验证的例子：

M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice-Hall, 1990.

第5章
Software Reliability Methods

形式化规约

“你要是像我一样熟悉时间，”帽匠说，“你就不会说浪费它。应该说他。”

刘易斯·卡洛尔《爱丽丝漫游奇境记》

很多软件项目都要经历几个月甚至几年的开发周期。在其生命周期中，软件产品会经常更新、增加新的功能或者调整以满足用户新的需求。一个完整的系统可能包含成百上千万行的代码，可能需要不同地域的成百上千人共同进行开发和维护。一个软件系统可能由很多部分组成，每一部分对应执行一些预先定义好的任务。不同部分之间的交互需要定义一些接口，正确地使用它们对于合并组件使之成为能够正常运行的系统非常重要。

形式化地描述设计并及时更新规约有助于保障项目在开发和维护过程中不同部分的一致性。通过明确定义每组代码的功能能够对编程的人力投入进行划分，而明确的接口规约则能够保障划分的有效性。每个小组的开发人员可以关注自己的部分，并根据相应的规约使用其他小组的工作。规约能够用来将设计和实现或者原始代码进行比对，并且可以作为实现代码变更的参考。

“一个程序是正确的”这句话，只有在给定一个规约的情况下才有意义。因此，形式化验证必须与具体的规约相关联。执行各类验证的工具均使用形式规约体系，并且能够检查系统初步规约的内部一致性。

形式化规约能够作为开发者和客户之间的契约。客户可以提出一些形式化的需求或者审核开发人员建议的规约。因此，规约能起到类似于法律合约的作用。当开发结束后，如果产品不能满足规约，客户能够对其提出异议，同样开发人员也可以用规约反驳并且表明发布的产品满足了规约。我们可以用验证工具来回答相关的功能实现是否满足了规约。

在本章我们将研究描述系统属性的形式化机制。描述系统本身与描述其属性是不同的行为，不能混淆。两种规约可能有时使用相似甚至相同的形式化机制。然而，在一些情况下，它们被表达为不同的形式化机制。例如，系统规约能够描述为程序或者状态机，它们能够描述系统行为的细节。另一方面，系统属性也能够用简明的逻辑符号描述。

5.1 规约机制的属性

规约需要是精确的，并且有唯一和一致的解释。因此，规约机制（specification formalism）需要良定义的语法和精确的语义。这样规约才是无二义的，并且问题“系统是否满足其规约?”才拥有唯一的答案。

通常在需求文档中使用的自然语言常常具有二义性，因此不适于形式化地描述软件。注意规约的不明确是指规约的解释可能不止一种方式。事实上，一个规约机制是无二义的并不意味着给定的属性不可以有多种描述的方式。虽然，如果一个形式化机制中对每个属性只有一种描述方法的话可能会带来一定好处，但这只是个小问题。规约机制并不一定需要文本形式。最近的趋势是使用可以进行可视化描述的形式化机制，这样的形式化机制能够给予所描述的系统一个更加直观的理解。可视化的机制将在第11章介绍。

规约机制需要直观并且易于使用，能够由程序员、系统设计者、需求工程师等撰写。它需要能够被其他人员理解，比如开发者和测试者。在形式化方法应用中，规约常常被算法使用，所以

它需要是机器可读的。然而，它的主要目的是在人与人之间传达信息。即使某个规约机制被用于测试和验证，如果不能反映人们所想表达的属性，它也是无用的。使用规约机制首先需要学习它。在实际使用中，学习新机制所花费的时间将会在减少调试时间上得到很好的补偿。

通常人们更愿意使用能够简明表示需求的规约机制。然而，如果是通过复杂的数学技巧来获得简明，反而导致得到的规约难以理解的话，那么这样的“简明”并不能带来任何好处。

如果一个好的规约机制能够检查或者验证一个系统和它的规约是否保持一致，那么我们称它是*有效的*（effective）。为了帮助测试和仿真，规约应该仅仅针对实际实验中能观察到的对象。一个更好的规约机制则可以用一个*高效的*（efficient）自动化方法来对其执行测试或者验证。人们想要进行的各种检查将会影响如何正确选择规约机制，例如：

- 检查规约集合中不包含矛盾。例如，规约的某个部分可能指定任务 A 必须在任务 B 之前执行，而另一部分指定任务 B 必须在任务 A 之前执行。需要注意：不存在一个执行能够满足一个有矛盾的规约。
- 检查规约是可实现的。例如，如果一个规约要求系统在给定时间内响应，而某些附加的时间约束在现实中无法实现，那么这样的规约就需要去除。
- 检查所要求的规约实现的正确性，或者至少使用充分的测试用例集合去测试它。

规约机制的*表达能力*是另一个重要的因素。显而易见，人们更愿意使用能表达广泛属性的机制。这里同样存在权衡问题。一个有更强表达能力的机制能够处理更多的规约实例。因此，这可能需要更多复杂的自动化验证算法去检查规约的一致性或者检查系统是否违背了某个规约。在一些情况下，选择一个有更强表达能力的机制会造成完全丧失使用某个算法去执行各种验证的能力。而进一步强的表达能力要求可能会导致其完全无法用手工来验证。因此，研究不仅仅需要找到有更强表达能力的规约机制，同样需要找到一个表达能力稍弱但更有效的规约机制。

规约有时能够被用于生成一些初始版本的代码。一些算法和工具能够自动地将形式化规约转换为模板代码以便在后期由程序员具体实现为实际代码，例如协议的通信模式就属于这类模板。这样的模板仍然需要在实际程序成型之前完善和修改。另一种相似的方法允许逐步从规约精化到实际程序。在每一步精化中会产生一个更加详细的规约，并且对精化步骤之间的一致性可以进行测试或者验证。

这样的将规约转换为代码的方法显然需要占用开发时间，以使规约的过程融入实现的任务中。而相应的由规约自动化生成的代码会更加可靠。

不幸的是，不存在一个机制可以完美地适用于任何情况。在一些情况下，很难选择一个非常适合的机制以满足规约的需要。正确的选择通常是不同机制的结合。由于设计出满足所有需求的规约机制很难，因此，就像存在很多不同的编程语言一样，同样存在很多不同的规约机制。主流或者已经成为标准的规约机制显然具有优势。在本章的后续部分，我们关注两个拥有大量使用者的规约机制：*线性时序逻辑*和*自动机*。在本书中，我们更多地关注描述系统的*动态*行为，也就是系统执行中的演进方式，而较少关注描述系统的静态属性。某些如一阶逻辑（在第 3 章介绍）和 Z 表示法 [35] 等规约机制更多地关注静态属性。

5.2 线性时序逻辑

在第 3 章讨论的一阶逻辑及命题逻辑能够表达状态的属性，每一个公式代表一个满足它的状态集合。因此，此类逻辑公式能够表达如初始条件、终止状态的断言或者*不变量*等属性。通过保存程序变量的两个副本，我们也能够表示初始状态和终止状态间的关系。然而这样的逻辑是*静态的*，因为它们表达的是一组状态，而不是它们在程序运行中的动态演变。

模态逻辑（modal logic）（参见 [71]）通过允许描述执行过程中不同状态间的关系对静态逻辑进

行了扩展。模态逻辑非常适合描述交互、并发或分布式系统中的断言，在这些系统中，不仅需要关注程序开始和结束时代码取值的变化关系，而且要关注执行过程中其他与状态顺序有关的属性。

线性时序逻辑（LTL）[91]是模态逻辑的一个实例。LTL一般通过对程序建模来描述交错序列[119]的属性，例如在4.10节中所给出的例子。LTL在静态逻辑$\mathcal{U}$的基础之上定义。逻辑$\mathcal{U}$能够描述状态的属性，但是不能描述不同状态间的变化。在本书中，我们将用一阶命题逻辑作为$\mathcal{U}$的一个示例。（为了简化起见，我们并不会明确提及在逻辑$\mathcal{U}$中使用的一阶结构$\mathcal{S}$。相应地，我们用符号$\models$代替$\models^{\mathcal{S}}$。）LTL的语法如下所示：

- $\mathcal{U}$中的每个公式都是一个LTL公式。
- 如果φ和ψ是公式，则$(\neg\varphi)$，$(\varphi\wedge\psi)$，$(\varphi\vee\psi)$，$(\bigcirc\varphi)$，$(\Diamond\varphi)$，$(\Box\varphi)$，$(\varphi\,\mathcal{U}\,\psi)$，$(\varphi\,\mathcal{V}\,\psi)$也是公式。

LTL公式一般通过一个无限状态序列$x_0x_1x_2\cdots$来解释，我们用ξ^k表示以x_k开始的序列$\xi=x_0x_1x_2\cdots$的后缀序列，也就是序列$x_kx_{k+1}x_{k+2}\cdots$。这样我们可以方便地按照如下的方式对任意序列ξ的后缀序列ξ^k定义其LTL语义。

- $\xi^k\models\eta$仅当$x_k\models\eta$，其中η是静态逻辑$\mathcal{U}$的公式。
- $\xi^k\models(\neg\varphi)$仅当$\xi^k\models\varphi$不成立。
- $\xi^k\models(\varphi\wedge\psi)$仅当$\xi^k\models\varphi$并且$\xi^k\models\psi$。
- $\xi^k\models(\varphi\vee\psi)$仅当$\xi^k\models\varphi$或者$\xi^k\models\psi$。
- $\xi^k\models(\bigcirc\varphi)$仅当$\xi^{k+1}\models\varphi$。
- $\xi^k\models(\Diamond\psi)$仅当存在$i\geqslant k$满足$\xi^i\models\psi$。
- $\xi^k\models(\Box\varphi)$仅当对于每个$i\geqslant k$，均有$\xi^i\models\eta$。
- $\xi^k\models(\varphi\,\mathcal{U}\,\psi)$仅当存在$i\geqslant k$有$\xi^k\models\psi$，并且对于所有$j$，当$k\leqslant j<i$，有$\xi^j\models\varphi$。
- $\xi^k\models(\varphi\,\mathcal{V}\,\psi)$仅当对于每个$i\geqslant k$，有$\xi^i\models\psi$，或者对于某个$j\geqslant k$，有$\xi^j\models\varphi$，并且对于每个$i$，当$k\leqslant i\leqslant j$，满足$\xi^i\models\psi$。

第一行的语义定义说明了底层逻辑$\mathcal{U}$的公式，即一个没有包含模态运算符的LTL公式在序列$(x_kx_{k+1}x_{k+2}\cdots)$的第一个状态$(x_k)$上进行解释。后三行是常见布尔非（“¬”）、合取（“∧”）和析取（“∨”）的描述。其余的定义则对应处理模态运算符。

模态运算符“○”称为“下一时刻”（nexttime）。当φ在从下一个状态x_{k+1}开始的后缀序列$x_{k+1}x_{k+2}\cdots$中成立，则公式$\bigcirc\varphi$在序列$x_kx_{k+1}x_{k+2}\cdots$中成立。类似地，如果φ在序列$x_{k+2}x_{k+3}\cdots$中成立，则$\bigcirc\bigcirc\varphi$成立。模态运算符“◇”称为“终将”（eventually）。当存在ξ的一个后缀中有φ成立，则公式$\Diamond\varphi$在序列ξ中成立。模态运算符“□”称为“总是”（always）。当φ在每个ξ的后缀中都成立，则$\Box\varphi$在序列ξ中成立。

我们能够构造结合了不同模态运算符的公式。例如，对每一个ξ的后缀ξ'，$\Diamond\varphi$成立，则公式$\Box\Diamond\varphi$在序列ξ中成立。因此存在一个ξ'的后缀ξ''满足φ。换句话说，无论我们在序列ξ中进行到多远（产生一个后缀ξ'），这里仍存在一个更后的后缀（ξ''）使φ成立。这意味着存在无限多个ξ的后缀使φ成立，或者换句话说，φ在ξ中“无限经常”（infinitely often）成立。现在考虑公式$\Diamond\Box\varphi$。当存在ξ的某个后缀ξ'满足$\Box\varphi$时，该公式在序列ξ中成立。那么φ对每个ξ'的后缀都成立。换句话说，φ从某个ξ的后缀开始到最后永远成立。

在此基础上，我们定义了两个额外的模态运算符“$\mathcal{U}$”和“$\mathcal{V}$”。运算符“$\mathcal{U}$”称为“直到”（until）。直观上，$\varphi\,\mathcal{U}\,\psi$断言$\varphi$成立直到某点（即后缀）使$\psi$成立。我们把“◇”看做是“$\mathcal{U}$”的特例，因为$\Diamond\varphi=true\,\mathcal{U}\,\varphi$。运算符“$\mathcal{V}$”称为“释放”（release）。直观上，当$\psi$永远满足（例如，对每个$\xi$的后缀）或者$\psi$保持满足直到（后缀上）某个点使$\varphi$和$\psi$都成立，那么$\varphi\,\mathcal{V}\,\psi$对于序列$\xi$成立。因此，$\varphi$“释放”了$\psi$将要在后面成立的需求（虽然没有要求$\psi$必须不能在后面的后缀中成立）。同样，“□”是“$\mathcal{V}$”的特例，因为$\Box\varphi=false\,\mathcal{V}\,\varphi$。

当 $\xi^0 \models \varphi$，即完整序列 ξ 满足 φ 时，我们省略后缀的上标，记为 $\xi \models \varphi$。符号 $\xi \models \varphi$ 代表序列 ξ 满足 LTL 公式 φ。相应地，$\xi \not\models \varphi$ 代表 ξ 不满足 φ。根据其语义定义，$\xi \not\models \varphi$ 仅当 $\xi \models \neg \varphi$。因此，对每个序列 ξ 我们有 $\xi \models true$ 和 $\xi \not\models false$。

我们经常会在 LTL 规约中省略括号。因此，我们会假定如下优先级："○"有最高优先级，其次"◇"、"□"、"U"和"V"优先级依次降低。二元运算符"U"向右靠齐，即 $\varphi \mathrm{U} \psi \mathrm{U} \eta = (\varphi \mathrm{U} (\psi \mathrm{U} \eta))$，而"V"运算符向左靠齐。我们有如下的连接规则：

$$\varphi \mathrm{V} \psi = \neg((\neg\varphi) \wedge (\neg\psi))$$
$$\varphi \rightarrow \psi = (\neg\varphi) \mathrm{V} \psi$$
$$\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$$
$$\Diamond \varphi = true \mathrm{U} \varphi$$
$$\Box \varphi = \neg \Diamond \neg \varphi$$
$$\varphi \mathrm{V} \psi = \neg((\neg\varphi) \mathrm{U} (\neg\psi))$$

相应地，可以仅使用模态运算符"○"、"U"和布尔运算符"∧"、"¬"定义更简单的时序逻辑。然而使用其他的运算符可以提高公式的可读性。

令 P 为一个允许多重操作的系统。这样的系统可以被描述为诸如一个转换系统或者自动机。P 中的每个执行可以通过一个状态序列来表示（见 4.10 节）。我们使用符号并且由 P 表示系统 P 生成的序列集合。我们用 $P \models \varphi$ 表示被描述为状态序列的 P 中的所有执行满足 φ，用 $P \not\models \varphi$ 表示不是 P 中的所有序列都满足 $P \models \varphi$。注意，并不总是当 $P \not\models \varphi$ 时都有 $P \models \neg\varphi$；事实上，有可能不是所有的序列满足 φ，但是同时其中的某些序列满足 φ。

举例说明，图 5.1 展示了一个简单的弹簧模型。我们能够拉这个弹簧然后释放掉。拉弹簧之后，弹簧可能失去弹性、保持伸长的状态或者恢复到原来的形状。这个系统有三个状态，s_1，s_2 和 s_3。其中 s_1 是初始状态（标记为没有与其他节点连接的进入箭头）。这个系统足够简单，可以用底层逻辑 $\mathcal{U}$ 来描述。相应地，每个状态被标记为集合 $AP = \{extended, malfunction\}$ 中的命题。状态 s_1 没有用任何上述命题标记，因此 $s_1 \models \neg extended \wedge \neg malfunction$。状态 s_2 只被标记为 *extended*，因此 $s_2 \models extended \wedge \neg malfunction$。最后 s_3 被标记为 *extended* 和 *malfunction*，即 $s_3 \models extended \wedge malfunction$。

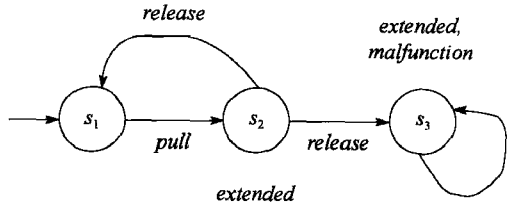


图 5.1 一个弹簧模型

这个系统拥有无限数目的序列，例如：

$$\xi_0 = s_1 s_2 s_1 s_2 s_1 s_2 s_1 \cdots$$
$$\xi_1 = s_1 s_2 s_3 s_3 s_3 s_3 \cdots$$
$$\xi_2 = s_1 s_2 s_1 s_2 s_3 s_3 s_3 \cdots$$

作为示例，让我们来思考序列 ξ_2 是否满足下列 LTL 公式。

- $\xi_2 \not\models extended$。公式 *extended* 并没有使用任何时序运算符。因此在序列 ξ_2 的第一个状态也就是 s_1 上对它进行解释。这个状态没有被标记为 *extended*，因此公式 *extended* 在 s_1 和序列 ξ_2 中不成立。
- $\xi_2 \models \bigcirc extended$。下一时刻运算符○在公式中被用来断言序列中的第二个状态，也就是 s_2 满足（即被标记为）命题 *extended*。
- $\xi_2 \not\models \bigcirc\bigcirc extended$。在 $\bigcirc\bigcirc extended$ 中使用了两次下一时刻（读作"下下个时刻会伸长"）断言第三个状态，也就是第一个状态后继的后继，被标记为 *extended*。然而这个状态 s_1 没有被标记为 *extended*。

$\xi_2 \models \Diamond\, extended$。公式$\Diamond\, extended$读作“终将伸长”，断言序列中存在某个状态满足 *extended*。确实，序列中的第二个状态被标记为 *extended*。

$\xi_2 \not\models \Box extended$。公式$\Box extended$读作“总是会伸长”，断言序列中的每个状态都满足 *extended*。（形式化地，它断言每个后缀序列的第一个状态都满足 *extended*。）此 LTL 公式对 ξ_2 不能成立，因为第一个和第三个状态都没有被标记为 *extended*。

$\xi_2 \models \Diamond\Box extended$。公式$\Diamond\Box extended$读作“最终总是会伸长”，断言序列中存在某个状态，其所有后续的状态都被标记为 *extended*。它从序列 ξ_2 的第四个状态（也就是 s_2）出现并且之后一直满足。

$\xi_2 \not\models (\neg\, extended)\, \mathcal{U}\, malfunction$。公式$(\neg\, extended)\, \mathcal{U}\, malfunction$读作“直到失效才能伸长”。如果它要在 ξ_2 中满足，则弹簧必须直到弹性失效才能伸长。对应其语义的定义，首先，为了让公式能够成立，在 ξ_2 中必须有一个状态满足 *malfunction*。其次，所有先前的状态必须满足$\neg\, extended$。ξ_2 中的第一个满足弹性失效的状态是第五个状态，也就是 s_3。检查之前的四个状态，我们发现并非所有的状态都满足$\neg\, extended$。特别是 ξ_2 中的第二个状态，也就是 s_2，不满足$\neg\, extended$。

刚才我们看到的几个时序公式的例子是在单个序列上加以解释。我们现在讨论对于一个系统 P 的时序公式的例子。回顾一下，$P \models \varphi$ 成立仅当对于 P 的每一次执行 ξ 都有 $\xi \models \varphi$。现在让我们来看以下对弹簧模型 P 的执行序列进行断言的一些属性：

$P \models \Diamond\, extended$。每一个系统的执行都会到达弹簧伸长的状态。因为弹簧不会永远处在初始状态 s_1，所以它是成立的。注意，如果我们在 s_1 上添加自循环，这样的情况就可能发生改变。

$P \models \Box(\neg\, extended \rightarrow \bigcirc extended)$。在 P 中的每次执行的每一个状态中，如果弹簧没有伸长，那么它一定在下一状态伸长。它是成立的，因为只有在 s_1 状态我们有$\neg\, extended$。对应于弹簧模型，在 P 中的任意执行序列中，每一次 s_1 发生后都立刻有 s_2 发生。

$P \not\models \Diamond\Box extended$。公式$\Diamond\Box extended$断言我们最终将会到达弹簧永远保持伸长的状态。对某一个特定的序列，也就是我们之前标出的序列 ξ_0，它将不能成立。在这个序列中 *extended* 和$\neg\, extended$永远交替下去。

$P \not\models \neg\Diamond\Box extended$。公式$\neg\Diamond\Box extended$是上面公式的否定形式。为了表明为什么这个公式对于 P 不能成立，我们首先写一个等价形式的公式。我们能够利用“$\Box$”和“$\Diamond$”的二元性：$\neg\Diamond\varphi = \Box\neg\varphi$和$\neg\Box\varphi = \Diamond\neg\varphi$。因此，可以得到$\neg\Diamond\Box extended = \Box\Diamond\neg\, extended$。这个公式断言从每一个序列中的状态（对应于“$\Box$”），存在一些将来的状态（对应于“$\Diamond$”），包含可能的当前状态，有$\neg\, extended$成立。也就是说，存在无限多的状态使$\neg\, extended$成立。虽然对于序列 ξ_0 它是成立的，但是对于 P 中的其他所有最终弹簧永远保持伸长的序列都不能满足。所以公式$\Diamond\Box extended$和其否定形式在 P 中都不能满足。

$P \not\models \Box(extended \rightarrow \bigcirc\neg\, extended)$。公式$\Box(extended \rightarrow \bigcirc\neg\, extended)$断言在每一个弹簧伸长的状态后，存在一个直接的后继状态满足$\neg\, extended$。虽然序列 ξ_1 满足该属性，但是在所有其他序列中此属性均不满足，因为它们最终都卡在 s_3 状态。

练习 5.2.1 上述模型当然是对弹簧进行了非常抽象的描述，它忽略了当弹簧拉伸或者释放时的连续状态。事实上，它假设每次弹簧被拉伸，它到达同一状态 s_2。它还假设了在被拉伸后，弹簧不能继续被拉伸，只能释放。这当然不是弹簧行为的唯一的模型。考虑另一种模型，弹簧可能在拉伸后失去弹性。在这个模型中，对于状态 s_1 会有两个后继状态：一个是弹簧拉伸，但是其弹性被破坏。而另一个是其弹性没有被破坏。使用自动机描述这个替换的模型，并且检验上述属性在该模型中是否被满足。

我们在4.12节提到过，有时会使用公平约束来阻止系统不合理的执行[3]。例如在本例中，可以用一个约束来断言每个执行不能永远保持在强连通分量 $\{s_1, s_2\}$ 中。该约束排除了执行序列 ξ_0。在这种情况下，系统 P 可允许的执行将会相应的减少，因此可能满足更多的属性。例如，之前在 P 中不能满足的属性$\Diamond\Box extended$ 在添加此约束后就可以被满足。

在一些情况下，我们更关注系统模型上的状态转换而不是状态本身，或者两者都关注。相应地，我们可以在转换序列或者状态和转换间变化的序列上描述LTL公式。对于后者，假设我们在模型结构上增加了转换名集合 TN（从 AP 中的命题抽取）。每当来自底层逻辑$\mathcal{U}$的公式被允许时，可以扩展LTL的语法使其能够使用来自 TN 的命题。定义 $first(\xi)$ 为序列 ξ 的第一个转换，相应地，我们在LTL语义描述中添加下面的项：

- $\xi^k \models t$，其中 $t \in TN$，仅当 $first(\xi^k) = t$。

5.3 公理化LTL

下面的公理和证明规则为命题化线性时序逻辑提供了一套完整的公理系统[88]。它们能够被用于证明给定公式 φ 是有效的，也就是被每个序列满足（在给定的命题集合下）。

在这个证明系统下，我们假定所有释放运算符“V”都被去除，例如，利用等式 $\varphi \mathsf{V} \psi = \neg((\neg\varphi)\mathsf{U}(\neg\psi))$ 进行转换。

这个公理系统包含三个部分，第一部分由下述八个公理组成。

A1 $\neg\Diamond\mu \leftrightarrow \Box\neg\mu$

A2 $\Box(\mu \rightarrow \psi) \rightarrow (\Box\mu \rightarrow \Box\psi)$

A3 $\Box\mu \rightarrow (\mu \wedge \bigcirc\Box\mu)$

A4 $\bigcirc\neg\mu \leftrightarrow \neg\bigcirc\mu$

A5 $\bigcirc(\mu \rightarrow \psi) \rightarrow (\bigcirc\mu \rightarrow \bigcirc\psi)$

A6 $\Box(\mu \rightarrow \bigcirc\mu) \rightarrow (\mu \rightarrow \Box\mu)$

A7 $(\mu \mathsf{U} \psi) \leftrightarrow (\psi \vee (\mu \wedge (\bigcirc(\mu \mathsf{U} \psi))))$

A8 $(\mu \mathsf{U} \psi) \rightarrow \Diamond\psi$

第二部分由一个对命题逻辑充分完备的公理系统组成。在系统中，我们不仅允许用命题公式来实例化公理和证明规则中的模板变量，还能够用任意命题LTL公式进行替换。这部分公理允许我们证明重言式，如$\Box A \vee \neg\Box A$。

最终，证明系统也包含证明规则：

Gen（时序一般化）

$$\frac{\mu}{\Box\mu}$$

（注意这个证明规则不同于3.5节中介绍的一阶逻辑证明规则GEN。）

我们将在6.6节看到还存在另一种吸引人的方法以证明命题LTL公式的有效性。换句话说，存在一个算法能够自动进行检验。另外，经验表明用于表达软件属性的LTL规约通常相当简短。因此，命题化LTL尤其适合对有限状态系统的属性进行自动化验证。

使用一阶逻辑表达状态属性的一阶线性时序逻辑，能够表达更多的属性。但是，另一方面，我们也失去了可判定性和完备性[1, 137]。一阶线性时序逻辑通常在演绎验证中使用（见第7章）。

5.4 LTL规约示例

5.4.1 交通灯

考虑一个交通灯，它能够在绿色、黄色和红色之间变换。其底层逻辑$\mathcal{U}$在这种情况下是一个

命题逻辑。命题 re，ye 和 gr 分别对应交通灯的红色、黄色和绿色。交通灯的颜色按如下顺序变换：

$$green \rightarrow yellow \rightarrow red \rightarrow green$$

我们假定交通灯永远变换下去。

交通灯在任意时刻仅能点亮其中某一种灯，这是系统的一个不变量，并且能够用如下 LTL 表达：

$$\Box(\neg(gr \wedge ye) \wedge \neg(ye \wedge re) \wedge \neg(re \wedge gr) \wedge (gr \vee ye \vee re))$$

当灯在绿色状态时，它将在变为黄色前一直保持绿色，这可以用 LTL 表达为：

$$\Box(gr \rightarrow gr \mathbin{U} ye)$$

因此，正确的灯的颜色变化被描述为：

$$\Box((gr \mathbin{U} ye) \vee (ye \mathbin{U} re) \vee (re \mathbin{U} gr)) \tag{5.1}$$

假设交通灯具有新的规则，现在红灯和绿灯之间也添加一个黄灯（给驾驶员的信号是“做好准备”）。由于上述规约不允许这样的情况，所以需要对它进行修改。

首先，我们尝试将规约修改为：

$$\Box(((gr \vee re) \mathbin{U} ye) \vee (ye \mathbin{U} (gr \vee re)))$$

这个规约是不正确的。$(gr \vee re) \mathbin{U} ye$ 允许在变为黄灯前等能够在绿灯和红灯间选择多次。此外，这个规约允许从绿灯变换到黄灯然后再次变为绿灯。

一个正确的规约应该是：

$$\begin{aligned}\Box\,&((gr \rightarrow (gr \mathbin{U} (ye \wedge (ye \mathbin{U} re))))\\ &\wedge (re \rightarrow (re \mathbin{U} (ye \wedge (ye \mathbin{U} gr))))\\ &\wedge (ye \rightarrow (ye \mathbin{U} (gr \vee re))))\end{aligned} \tag{5.2}$$

第一行允许按 $green \rightarrow yellow \rightarrow red$ 这样的顺序。第二行允许按 $red \rightarrow yellow \rightarrow green$ 的顺序。虽然式（5.2）的前两行对应的状态都有黄灯亮，但是它们仅仅处理绿灯或者红灯亮后黄灯亮的情况。它们不会提供以黄灯开始的情况下交通灯的行为信息。因此，我们添加了第三行。

目前给定的交通灯规约允许它从任意的颜色开始。如果我们想要指定初始颜色是红色，我们仅需要添加一个 re 的合取。注意，如果交通灯必须从红灯开始，那么我们将不需要式（5.2）的第三行。

5.4.2 顺序程序的属性

假设一个程序被建模为如 4.4 节中的转换系统，并且底层逻辑 $\mathcal{U}$ 可以用于定义转换系统。考虑如下新增的符号：

en_α 转换 α 的可执行条件。

en_{P_i} 至少 P_i 中的一个转换可执行。如果 P_i 也表示为同名进程的转换集合，那么它能够被表示为 $\bigvee_{\alpha \in P_i} en_\alpha$。

$init$ 当前状态是一个初始状态。在本书中，我们经常用 Θ 表示转换系统的初始条件。

$finish$ 当前状态是一个终止状态。如果 T 是所有转换的集合，这里就能够表示为 $\bigwedge_{\alpha \in T} \neg en_\alpha$。（回顾之前我们假定所有执行序列都是无限序列，因此，我们可以通过对最终状态进行无限重复来扩展有限序列。）

在下面的属性中，我们将使用基于底层逻辑 $\mathcal{U}$ 表达的 ψ 作为状态断言。

部分正确性，与初始条件和最终断言 ψ 相关。

$$init \wedge \Box(finish \rightarrow \psi)$$

它断言程序将由满足 *init* 的状态开始，并且如果到达终止状态，则这个状态满足 ψ。

终止性，与初始条件相关。

$$init \wedge \Diamond finish$$

它断言程序将由满足 *init* 的状态开始并且最后到达终止状态。

完全正确性，与初始条件和最终断言 ψ 相关。

$$init \wedge \Diamond(finish \wedge \psi)$$

它断言程序将由满足 *init* 的状态开始并且最终终止在满足 ψ 的状态。

程序开始满足初始条件 *init* 并且 ψ 是一个不变量，也就是在系统执行过程中总是成立。

$$init \wedge \Box\psi$$

5.4.3 互斥

我们在 4.6 节已经讨论过互斥问题。考虑一对进程 P_1 和 P_2 连接到一个共享设备，如打印机。它们不能同时使用设备，因为从两个进程同时打印会导致混乱的结果。因此，会利用一个特殊的机制来对它们进行控制。在该机制中，每个进程必须在代码中进入一个特殊的区域，称为临界区。一个进程只有在临界区内才能进行打印。为了防止同时打印，进程使用相应的协议实现互斥，即确保不可能同时两个进程都进入临界区。作为协议的一部分，在进入临界区之前，每个进程进入一个尝试区（trying section），在这个区域内表明自己进入临界区的目的。互斥协议随后考虑各种情况，允许一个进程进入临界区，或者让它停在那里，在进入临界区之前一直在尝试区等待。可以在这样的协议上添加一些需求。首先，让我们定义一些能用于描述这些需求的命题。

$tryCS_i$　进程 P_i 进入它的尝试区，也就是试图进入它的临界区。

$inCS_i$　进程 P_i 在它的临界区 CS_i 中。

利用以上命题，我们定义的第一个属性是互斥性：在任何时间里，只有一个进程能够在它的临界区中。

$$\Box\neg(inCS_1 \wedge inCS_2)$$

另一个是关于响应能力（responsiveness）的属性。我们要求每个尝试进入临界区的进程在最终都能被允许进入。

$$\Box(tryCS_i \rightarrow \Diamond inCS_i)$$

为了使互斥协议能够正确地执行，我们可能进一步要求：当一个进程进入它的尝试区时，它将停留在那里，除非它进入临界区。

$$\Box(tryCS_i \rightarrow ((tryCS_i \cup inCS_i) \vee \Box tryCS_i))$$

5.4.4 公平性条件

现在我们将使用 LTL 公式来表示在 4.12 节中提到的一些公平性条件。其中，我们将会使用以下的转换命题。

$exec_\alpha$　转换 α 被执行。这是一个既包含了状态又包含了转换的例子，其中 α 是转换名。如 5.2 节结束时所述，表达这个命题需要使用标记在转换上的命题（或者等价地，在状态上增加关于下一个执行的转换信息）。

$exec_{P_i}$　P_i 中的一个转换被执行。这是 $\bigvee_{\alpha\in P_i} exec_\alpha$ 的简写。

弱转换公平性。

$$WTF = \bigwedge_{\alpha\in T} \neg\Diamond\Box(en_\alpha \wedge \neg exec_\alpha)$$

它同样可以用下面的等价方式表达：

$$\bigwedge_{a\in T}(\Diamond\Box en_a \rightarrow \Box\Diamond exec_a)$$

强转换公平性。

$$STF = \bigwedge_{a\in T}(\Box\Diamond en_a \rightarrow \Box\Diamond exec_a)$$

弱进程公平性。

$$WPF = \bigwedge_{P_i} \neg\Diamond\Box(en_{P_i} \wedge \neg exec_{P_i})$$

它同样可以用下面的等价方式表达：

$$\bigwedge_{P_i}(\Diamond\Box en_{P_i} \rightarrow \Box\Diamond exec_{P_i})$$

强进程公平性。

$$SPF = \bigwedge_{P_i}(\Box\Diamond en_{P_i} \rightarrow \Box\Diamond exec_{P_i})$$

在图 4.8 中阐述的公平条件之间的关系现在能够根据逻辑蕴含进行阐述。回顾一下 $\varphi\rightarrow\psi$ 的意思是 φ 强于 ψ，因此我们有 $STF\rightarrow SPF$，$SPF\rightarrow WPF$ 和 $STF\rightarrow WTF$。

练习 5.4.1 考虑如下的交通灯规约（5.2）的另一种规约。

$$\begin{aligned}&\Box((gr\rightarrow(gr\ \mathrm{U}\ (ye\ \mathrm{U}\ re)))\\&\quad\wedge(re\rightarrow(re\ \mathrm{U}\ (ye\ \mathrm{U}\ gr)))\\&\quad\wedge(ye\rightarrow(ye\ \mathrm{U}\ (ye\vee re))))\end{aligned}\tag{5.3}$$

解释一下为什么这个规约不能充分捕获信号灯之间顺序的描述。（提示：注意直到“U”的精确定义。）

5.5 无限字上的自动机

自动机理论在计算机科学中扮演着重要的角色。各种类型的自动机被用于编译、自然语言分析、复杂性理论和硬件设计等领域。同样，自动机理论也非常适合领域建模和系统验证。

有限自动机主要是在有限转换系统上的状态机。使用转换系统对程序进行建模，并且使用自动机描述程序属性具有很大的好处：这意味着可以使用同一种表达方式描述程序和它的规约。随后可以利用图算法执行自动验证，我们将在第 6 章进一步阐述。

在无限字上的有限自动机，称作 ω 自动机，能够描述程序的执行。系统的属性也可以用 ω 自动机来描述或者翻译，例如，从 LTL 转换成这样的自动机，我们将在 6.8 节进行介绍。

建模后的系统能够呈现出无限和有限行为。就像我们已经讨论过的，为了避免处理这两种行为，我们可以将有限行为转换为无限行为。这可以通过使用一个特定的 *no_op* 转换来实现，它不改变系统的状态并且仅当所有其他转换不可执行的时候才可执行。因此，我们可以仅关注在无限字上的有限自动机，即 Σ^{ω} 中的字，其中 Σ 是一个有限字母表，上标 ω 表示无限次数的迭代。能够被 ω 自动机接受的语言称为 ω 正则语言。每个这样的语言能够用字母表 Σ 中的字母及运算符+（结合）、.（联结）、*（表示 0 或多次重复）和 ω（无限多次重复）构成的表达式来表达。

我们在第 4 章讨论过将系统行为建模为状态序列、转换序列或者包含两者的交替序列的各种可能性。为了与之前的选择保持一致，我们通常将系统行为建模为状态序列。相应地，我们将会定义能够识别状态序列的自动机。由于自动机通常在转换上而不是在状态上添加标签，我们定义的自动机可能看上去不那么传统。但是这样的定义也有其优点，例如将在 6.8 节描述的从 LTL 到自动机的转换将会更为直接。同样，我们也可以通过对状态和转换均进行标记来扩展模型，这将使得我们的自动机可以识别状态和转换交替的序列。由于此类扩展简单明确，因此相关

细节在此不再赘述。

人们应该考虑到 ω 自动机并不是真正意义上实际地识别执行，由于执行是无限的，甚至识别其中一个都会无法终止。然而，ω 自动机提供了一个有限的方式来表示无限的执行，并且拥有一个有限的结构。这样的有限性使得设计自动验证算法成为可能。

无限字的 ω 自动机中最简单的一类是 Büchi 自动机 [22]。在此，我们将会描述它的一个变体，其中标签在状态上而不是转换上定义。一个 Büchi 自动机 $\mathcal{A}$ 是一个六元组 $\langle \Sigma, S, \Delta, I, L, F \rangle$，其中：

- Σ 是有限字母表。
- S 是有限状态集合。
- $\Delta \subseteq S \times S$ 是转换关系。
- $I \subseteq S$ 是开始状态集合。
- $L : S \to \Sigma$ 是状态标签的映射函数。
- $F \subseteq S$ 是接受状态集合。

在 Büchi 自动机的图表示中，我们用没有连接其他任何节点的进入箭头标记一个初始状态。我们使用双圆标记接受状态。在如图 5.2 所示的自动机例子中，$\Sigma = \{\alpha, \beta\}$，$S = \{s_1, s_2\}$，$I = \{s_1, s_2\}$，例子中的标签是 $L(s_1) = \alpha$ 和 $L(s_2) = \beta$ 并且 $F = \{s_1\}$。

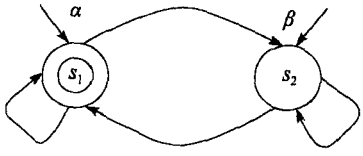


图 5.2 一个 Büchi 自动机

$\mathcal{A}$ 在 v 上的一个运行 ρ 对应于在自动机图中从初始状态开始的一个无限路径，其中在该路径上的节点被标记为对应 v 中的字母。我们称 v 是自动机 $\mathcal{A}$ 的一个输入，或者说 $\mathcal{A}$ 读取了 v。从形式化的角度，令 v 是 Σ^ω 上的一个字（字符串，序列）。我们还能够将 v 表达为函数 $v : Nat \to \Sigma$，即 $v = v(0)v(1)v(2)\cdots$。$\mathcal{A}$ 在 v 上的一个运行是一个映射函数 $\rho : Nat \to S$，其中：

- $\rho(0) \in I$，第一个状态是初始状态。
- 对于 $i \geqslant 0$，$(\rho(i), \rho(i+1)) \in \Delta$。运行中的第 i 个状态 $\rho(i)$ 到第 $i+1$ 个状态 $\rho(i+1)$ 的跳转符合转换关系 Δ。
- v 上第 i 个元素，也就是 $v(i)$ 和状态 $\rho(i)$ 的标签 $L(\rho(i))$ 是相同的，也就是 $v(i) = L(\rho(i))$。

令 $inf(\rho)$ 表示在运行 ρ（将运行看做一个无限路径）中无限经常出现的状态集合。注意，$inf(\rho)$ 是一个有限集合。当 $inf(\rho) \cap F \neq \varnothing$，也就是当某个接受状态在 ρ 中无限经常出现时，Büchi 自动机 $\mathcal{A}$ 在无限字上的运行 ρ 是可接受的。

Büchi 自动机 $\mathcal{A}$ 上的语言 $\mathcal{L}(\mathcal{A}) \subseteq \Sigma^\omega$ 由所有 $\mathcal{A}$ 接受的字组成。对于图 5.2 所示的在 $\Sigma = \{\alpha, \beta\}$ 上的自动机，考虑仅含有 α 的无限字 α^ω。在这个字上的自动机 $\mathcal{A}$ 的一个运行必须由状态 s_1 开始，因为 s_1 是其中仅有的一个标记了 α 的状态。然后继续运行，永远通过 s_1。字 α^ω 是可被自动机 $\mathcal{A}$ 接受的，并且因此 $\alpha^\omega \in \mathcal{L}(\mathcal{A})$。这是因为状态 s_1 是接受状态并且无限经常出现。

考虑字 β^ω，这个字由无限多个 β 组成。在这个字上自动机的（唯一）运行开始于并且保持在状态 s_2。这不是一个可接受的运行，因为没有出现无限多次的接受状态（事实上，唯一的接受状态 s_1 根本没有出现）。因此，$\beta^\omega \notin \mathcal{L}(\mathcal{A})$。最后，考虑包含 α 和 β 之间交替的字，记做 $(\alpha\beta)^\omega$。其对应的运行由 s_1 开始并且在 s_2 和 s_1 之间永远交替下去。既然 s_1 是一个接受状态并且出现无限多次，所以这个字在 $\mathcal{L}(\mathcal{A})$ 中。

在图 5.2 中自动机的语言能够用 ω 正则表达式 $(\beta^* \alpha)^\omega$ 来表示。这个记号描述了如下的无限重复：一个有限（可能是空的）β 序列，跟随一个 α。因此，当一个在 $\Sigma = \{\alpha, \beta\}$ 上的无限字包含无限多次 α 时，它能被这个自动机接受。如上所示，这包含了字 $(\alpha\beta)^\omega$，也就是一个由 α 开始的在 α 和 β 之间的无限交替。自动机 $\mathcal{A}$ 同样接受字 α^ω 和 $\beta\alpha\beta^2\alpha\beta^3\alpha\cdots$。

5.6　使用 Büchi 自动机作为规约

在上文我们提到，使用自动机的好处之一是可以使用同一种方式来表示被建模系统及其属性。我们能够使用 Büchi 自动机 $\mathcal{A}=\langle 2^{AP}, S, \Delta, I, L, F\rangle$ 表示一个系统。它包含状态集合 S。状态间的直接后继关系为 $\Delta\subseteq S\times S$，即 $(s, r)\in\Delta$，其中 r 可以由 s 通过执行一个原子转换获得。初始状态集合为 $I\subseteq S$。

标签函数 $L: S\to 2^{AP}$ 为每个状态标注一个来自有限集合 AP 的命题集合。这些命题在状态中成立，而 AP 中的其他命题在此状态中不能被满足（见 4.8 节）。

对于表示系统的自动机，我们通常设置所有状态 S 都是接受状态。事实上，我们仅在建模系统加入了一些公平条件时才区分接受和非接受状态 [3]。因此，接受状态集合有时被称为系统的公平性条件[⊖]。

当从 Σ 给定下一个输入字母，对下一个状态存在选择时，Büchi 自动机是非确定的。也就是说，存在两个状态 $r_1, r_2\in S$ 使得 $L(r_1)=L(r_2)$，并且 $r_1, r_2\in I$，或者从同一状态 $s\in S$ 存在至少两个转换 $(s, r_1), (s, r_2)\in\Delta$。非确定 Büchi 自动机对给定输入可能存在不止一个的执行。注意对应于接受的定义，如果对 v 存在一个接受执行，那么 v 就被 $\mathcal{A}$ 的语言 $\mathcal{L}(\mathcal{A})$ 包含。

系统 $\mathcal{A}$ 的属性规约能够用在与 $\mathcal{A}$ 相同的字母表上构建的自动机 $\mathcal{B}$ 来描述。当系统 $\mathcal{A}$ 和规约 $\mathcal{B}$ 的语言之间存在包含关系也就是 $\mathcal{L}(\mathcal{A})\subseteq\mathcal{L}(\mathcal{B})$ 时，系统 $\mathcal{A}$ 满足规约 $\mathcal{B}$。因此，待建模系统的每一个行为必须包含在规约允许的行为中。这个简单的规约机制能够被用作自动验证的基础，我们将在下一章进一步阐述。

实际上，如果规约自动机中每个状态分别对应于命题变量 AP 中一个简单的赋值，那么其效率有时并不高。考虑在图 5.3 左边显示的自动机。它含有三个状态 r_1，r_2 和 r_3。这些状态有一个共同的后继和前驱状态 q。这种多个状态拥有共同的后继和前驱的情况是极其常见的。为了获得更小和更简单的自动机表示，我们将这些状态合并为一个状态，具体做法是替换共享相同的后继和前驱的状态（在本例中，r_1，r_2 和 r_3 分别对应赋值 $\{A\}$，$\{A, B\}$ 和 $\varnothing$）为一个被标记为满足这些赋值的命题公式的状态 r。状态 q 的标签 $\{B\}$ 被替换为对应命题公式 $(\neg A)\wedge B$。简化的自动机如图 5.3 右边所示。

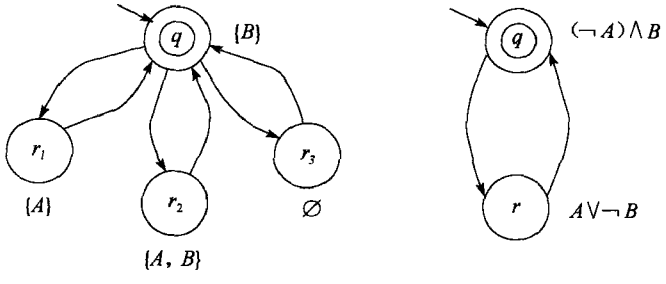


图 5.3　一个 Büchi 自动机的两种表示

相应地，我们需要改变自动机的定义，事实上唯一的改变是每个状态现在对应于 AP 上的一些赋值。因此，我们的标签函数变为 $L: S\to 2^{2^{AP}}$，其中 $2^{2^{AP}}$ 表示 AP 子集的集合。这样子集的集合能够使用一个命题公式表示。然后，每个状态 $s\in S$ 就可以映射到公式 φ_s，相应地，s 上可以添

⊖ 对于各种公平性条件，不足以在自动机上标注 Büchi 可接受条件（实际上，由于各类 ω 自动机之间的等价性，可以将此自动机转换到能表达原始系统的公平序列的另一种等价类）。我们能够使用含有更多结构的接受条件（例如，将在 6.4 节描述的广义 Büchi 条件）。

加的标签均满足 φ_s。在图 5.3 的例子中，公式 $A \vee (\neg B)$ 对应于状态 $\{\{A\}, \{A, B\}, \varnothing\}$。这些状态中的每一个都满足该公式，而状态 $\{A\}$ 不满足。注意：尽管自动机的定义改变了，但自动机可识别的语言并未扩展，而是提供了一种更简洁的表达方式。

我们需要重新定义这种自动机的运行。事实上，只需要对运行定义中的最后一个条件进行适当的改变即可，也就是输入字母与状态上的标签一致的要求。对输入字的字母表仍然是 $\Sigma = 2^{AP}$。在 $L(\rho(i))$ 表示命题公式的情况下，对应条件可以描述为：

$$v(i) \models L(P(i))$$

在这个扩展上下文中，非确定性也需要重新谨慎地定义。如上所述，当系统读取来自 Σ 中的输入字母时存在下一个状态间的选择，那么自动机是非确定的。在当存在两个状态 r_1，$r_2 \in S$ 有 $L(r_1) \wedge L(r_2) \neq false$（“$\neq$”在这里的含义是“非逻辑等”，也就是这个连接词不是一个矛盾），并且 r_1，$r_2 \in I$，或者从同一状态 $s \in S$ 存在至少两个转换 (s, r_1)，$(s, r_2) \in \Delta$ 时，就存在不确定性。对应于这样的定义，后面如图 5.5 所示的自动机是确定的，而如图 5.6 所示的自动机是非确定的。

当然，如果一个状态上标注的条件逻辑等价于 $false$，那么此状态能够被移除，相应的进边和出边也将一并删除。这是因为这个状态上的条件不能满足任何一个赋值，因此也不能参与到任何的运行中。此外，相关的简化包括移除那些不能从初始状态到达的节点，同时自然也包括它们的所有进边和出边。

Büchi 自动机规约示例

如图 5.4 所示的自动机描述了两个进程不能同时进入它们的临界区（CR_0 和 CR_1）的互斥属性。命题 $inCR_0$（$inCR_1$）标记进程 P_0（P_1）在它的临界区内时的状态。互斥需求可以使用 LTL 表示为 $\Box\neg(inCR_0 \wedge inCR_1)$。值得注意的是，在 Büchi 自动机中不存在标记为 $inCR_0 \wedge inCR_1$ 的状态。因此，对应于 Büchi 自动机执行的定义，不存在包含这样标签的执行。因此，也就自然不存在自动机能够接受并且能够到达状态 $inCR_0 \wedge inCR_1$ 的执行。

图 5.5 表示的是一个描述活性（liveness）属性的自动机。直观而言，活性属性描述的是某些最终将会满足的属性 [83]。图 5.5 中的属性断言一个进程将会最终进入它的临界区，用 LTL 表示为 $\Diamond inCR_0$。

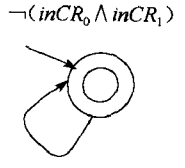


图 5.4 互斥

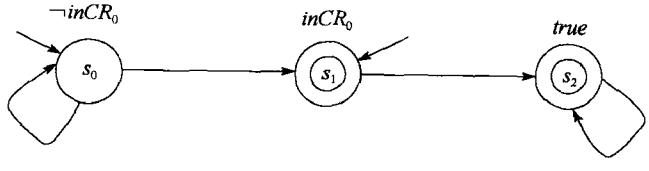


图 5.5 活性属性

5.7 确定性 Büchi 自动机

确定性 Büchi 自动机在概念上似乎比非确定性自动机简单。不过，存在没有与之等价的确定性自动机的非确定性 Büchi 自动机。

在 $AP = \{A\}$ 上无限字的语言包含了那些在其上 A 满足有限多次的字，这意味着被这个自动机接受的每个字都会有一个后缀之上没有状态满足 A。如图 5.6 所示的语言能够被非确定性 Büchi 自动机接受，其中状态 s_0 被标记为 $true$，或者等价地标记为 $A \vee \neg A$。因此，当自动机处于状态 s_0 并且下一个输入字母是 A 时，存在保持在 s_0 和转换到 s_1 两种选择。在输入 v 上，自动机保持自循环 s_0 直到某个点当它“猜测”将不再存在满足 A 的状态时。然后它将跳转到 s_1。由于

自动机是非确定的，因此必然存在一个正确的猜测从而接受 v。

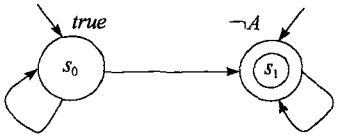


图 5.6　接受 A 满足有限多次的字的自动机

我们将要说明不存在一个确定性自动机能够识别这个语言 [141]。假设这里存在一个确定性 Büchi 自动机能够识别这个语言，那么它必须在读取一个有限字符串 $(\neg A)^{n_2}$ ($n_1 \geqslant 0$) 之后到达某个接受状态，否则字 $(\neg A)^{\omega}$ 将不能被接受。(回顾在确定性自动机中，对于每个输入字最多存在一个运行。) 从这个状态继续，确定性自动机必须在对某个 $n_2 \geqslant 0$ 的 $(\neg A)^{n_1} A(\neg A)^{n_2}$ 之后到达一个接受状态。因此，这个自动机将会接受包含无限多个满足 A 的状态的形如 $(\neg A)^{n_2} A(\neg A)^{n_1} A(\neg A)^{n_3} \cdots$ 的字，这就会形成矛盾。

当我们关注这个语言的补集时会发现一个有趣的现象，也就是包含无限多个 A 的无限字语言能够被一个确定性自动机识别。以图 5.2 所示的自动机为例，假设 $\alpha = A$，$\beta = \neg A$，就可以展示这个现象。因此，我们能够得到如下结论，能够被确定性 Büchi 自动机接受的语言集合在补操作下不封闭。

如果某个字母表 Σ 上的语言 $\mathcal{L}(\mathcal{A})$ 能被一个确定 Büchi 自动机 $\mathcal{A}$ 识别，那么它需要满足下述条件：

对于在 Σ 上的每个无限字 σ，σ 在 $\mathcal{L}(\mathcal{A})$ 中当且仅当 σ 中存在无限多的有限前缀，并且这些前缀可以对应到一个在 $\mathcal{A}$ 中可到达接受状态的运行。

以上条件源于这样的事实：在确定性的 Büchi 自动机中，对应每个字以及对应字的每个前缀，最多存在一个相应的执行。

5.8　其他规约机制

在第 6 章中我们将表明，每个命题 LTL 规约都能够被转换为对应的 Büchi 自动机。因此，Büchi 自动机的表达能力（能表述的规约或者语言的集合）至少与 LTL 相当。事实上，前者具有严格的、更强的表达能力。举例来说，LTL 不能表达 A 至少在偶数状态中（从 0 开始计数）成立的无限字语言 [147]。能够识别这个语言的 Büchi 自动机如图 5.7 所示。注意，我们没有对奇数状态强加任何的限制，因此 A 可能在一些奇数状态中成立。进一步地，如果需要 A 不能在任何奇数中成立，我们就能够简单地使用 LTL 表达这个语言。

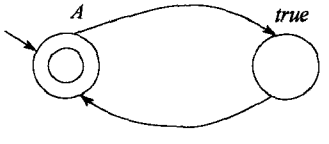


图 5.7　识别 A 在偶数位置成立的字的自动机

下面的规约机制与 Büchi 自动机具有同样的表达能力：

一元二阶逻辑[22]。这个逻辑能够在包含下述组件的数学结构上进行描述：

Nat　自然数集合。

Q_A，Q_B，…　一元关系，每个这样的关系定义了满足它的自然数子集。这样的关系 Q_A 与 LTL 命题 A 通过以下情况进行关联：当 A 在序列的第 i 个状态中成立时，$Q_A(i)$ 成立。

$<$　是自然数上通常的序关系。

succ　是自然数上的后继关系。

0　是表示最小自然数的常量。

值得注意的是，*succ* 和 0 能够通过使用 $<$ 关系消除。作为二阶逻辑，它允许在简单变量

和变量集合上使用量词。例如，$\exists X\varphi$ 是一个一元二阶逻辑公式，其中 X 是自然数变量的集合（然而 Q_A，Q_B，…是一元关系“常量”）。在 φ 中，我们可以使用 $y\in X$，其中 y 是一个简单变量（在自然数上）。相应地，A(至少）出现在所有偶数位置这一属性能够被表示为：

$$\exists X(0 \in X \wedge \forall y \forall z(succ(y,z) \rightarrow (y \in X \leftrightarrow \neg z \in X)) \wedge \forall y(y \in X \rightarrow Q_A(y)))$$

这个公式断言这里存在包含 0（记做 $0\in X$）的自然数 X 集合，并且对于每一个自然数对，其中的一个是另一个的后继，X 只包含其中一个元素。集合 X 包含在关系 Q_A 中（对应于命题 A）。它可被如图 5.7 所示的自动机的接受序列集合满足。

包含量词的 LTL。我们能够通过允许二阶量词来扩展 LTL。量化的变量能够被量词范围内的所有其他命题使用。上面的属性就能够用包含量词的 LTL 表示：

$$\exists X((X \wedge \Box(X \rightarrow \bigcirc \neg X) \wedge \Box(\neg X \rightarrow \bigcirc X)) \wedge \Box(X \rightarrow A))$$

ω 正则表达式。它们是对于无限迭代运算符“ω”扩展的正则表达式。上述属性能够使用 ω 正则表达式 $(A\Sigma)^\omega$ 表达，这里用 Σ 表示在 2^{AP} 上的任何状态，用 A 表示任何满足 A 的状态。

*一元一阶逻辑*可以由上面提到的一元二阶逻辑获得，它不允许集合变量（以及其上的量词）。它的表达能力等价于 LTL。

直观而言，Büchi 自动机使用其有限状态结构作为一个有限内存。需要无界内存的属性不能使用这个规约机制描述，并且也不能够使用表达能力更弱的 LTL 描述。虽然实际上限制程序为有限多个状态，但是算法的形式化描述通常在规约中使用无界结构。这通常能生成更加直观和简洁的规约。

考虑一下在某个有限域 M 上的 fifo（先进先出）消息传递。fifo 队列的抽象描述不会对缓冲区的大小进行限制。当然如果我们想要这么做，也可以添加这样的限制；一个给定的 fifo 队列的大小一般是一定的，基于这样的前提，理论上我们当然能够生成相应的有限状态的规约。这样的规约考虑了在 fifo 机制允许下并且在限制了消息数目的前提下所有可能的队列内容。然而这样将无法直观地描述那些不包含特定限制的抽象 fifo。

很多标准结构，如 fifo 队列，不能够直接使用 Büchi 自动机或者 LTL 公式表达 [133]。假设用命题 add_x 表示将一个元素 x 插入到队列中的操作，并且从队列的顶部移除一个元素的操作用 $remove_x$ 表示。在这种情况下，Büchi 自动机的有限内存容量不足以描述对应于 fifo 行为的 add_x 和 $remove_x$ 序列集合。

规约机制的表达能力不是我们唯一需要考虑的重要因素。我们也应该考虑检验给定系统（使用某种规约机制表达，如自动机）是否满足给定规约的复杂性。例如，虽然一元一阶或二阶逻辑分别与 LTL 或 Büchi 自动机具有同样的表达能力，但它们并不常用于模型检验。这是因为它们允许过于简洁的属性表示，以至于将表示翻译为自动机或者等价的 LTL 公式，可能导致 NONELEMENTARY（非初等函数）级爆炸 [134]。

对于给定的有限状态系统，检验一元一阶或二阶逻辑规约的复杂性在公式的大小上一般是 NONELEMENTARY。这可能无法使用这样简洁的规约机制用于模型检验。另一方面，对于一个有限状态系统检验 LTL 规约是 PSPACE 完全的（见 6.9 节）。然而，我们能够举例说明某些用一元二阶（一阶）逻辑公式描述的规约，其等价的 Büchi 自动机（LTL 公式）的最小规模已达到此一元二阶（一阶）逻辑大小的 NONELEMENTARY 级别。因此，我们能够说本质上高复杂度依赖于需要验证的属性而不是所使用的规约机制。此外，NONELEMENTARY 爆炸不是在任何一元一阶或者二阶逻辑属性的情况下都是如此。因此，如 MONA[41] 之类的系统，能够成功地验证用一元二阶逻辑表达的属性。

事实上，对于大多数的用途而言，使用有限的小 LTL 属性去描述大多数的硬件和软件的需求已经足够了。

5.9 复杂的规约

经验表明有少量简单属性是很多系统规约所共有的 [93]。另一方面，当然也存在一些特别的系统需要满足某些特别的、非标准的属性。

举一个典型案例：*描述功能交互*。一个软件系统通常提供一些功能。一个典型的例子是电话交换机，它需要处理很多不同的服务，例如*简单呼叫*、*免费呼叫*、*会议呼叫*等。当添加一个新功能时，要求它的行为不会对已有的功能产生非预期的影响。为如上所述的功能交互提供一个精确的规约是一项困难的挑战。

另一个例子涉及*遗留代码*的一致性。我们想要保持新开发代码和某些老代码的行为保持一致。老代码的规约可能不保存在文档中，并且可能仅仅被少数老程序员知道。

在这些例子中，使用规约机制描述系统需要的属性仅仅提供了部分解决方法。我们可能需要比较不同版本或者系统的不同部分，并且说明它们在行为上是一致的。比较不同系统的行为能够使用第 8 章介绍的进程代数的上下文实现。其中，不同的系统使用不同的标准进行比较。然而，处理如功能交互和遗留代码的一致性问题还没有得到完美的解决，并且对软件工程师形成实际的挑战。

5.10 规约的完整性

我们很清楚一个系统必须满足它的规约，然而很难确定是否已给定足够的规约。当设计一个新系统时，并不一定所有的重要需求都已经提出了。有时甚至在部署系统后，还有一些重要的约束没有考虑。在很多情况下，可以算法性地检验给定的需求集合内彼此会不会存在矛盾（见 6.11 节）。然而，检验约束集合是否完整，即是否包括了每个合理的规约，通常是不可行的，这是因为我们永远可能赋予系统新的规约。

作为一个不完整规约的例子，我们考虑一个系统，其输入为数组变量序列$A[1]$，…，$A[n]$，输出为上述变量的总和 *sum*。我们能够描述：在初始条件下，$A[1]$，…，$A[n]$ 以及 n 是自然数。在终止时，有 $sum = \sum_{i=1}^{n} A[i]$。

这是一个不完整的规约。把 0 赋值给所有数组变量和 *sum* 满足这个规约。而我们“忘记”提到的是数组变量在执行中不会改变。现在就完整了吗？还不完整，我们还需要规定 n 在执行中不会改变。

规约的完整性基本依赖于设计者的独创性。不过本章中已经提到了软件系统的一些“标准”属性，我们在建立需求集合时也可以参照比较。

规约练习

练习 5.10.1 给出 4.6 节中 Dijkstra 临时的互斥算法和 Dekker 互斥算法的 LTL 规约。

练习 5.10.2 描述一个电梯系统的属性，使用如下命题：

at_i 电梯在第 i 层

go_up 电梯上升

go_down 电梯下降

$between_i$ 电梯在第 i 层和第 $i+1$ 层之间

stop 电梯没有移动

open 电梯门是打开的

$press\_up_i$ 某个人在第 i 层按下向上按钮

$memory\_up_i$ 电梯“记下”在第 i 层向上按钮被按下

$press\_down_i$ 某个人在第 i 层按下向下按钮

$memory\_down_i$ 电梯“记下”在第 i 层向下按钮被按下

$press_i$ 某个人在电梯内按下了去第 i 层的按钮

$memory\_press_i$ 电梯“记下”某个人在电梯内按下了去第 i 层的按钮

alarm 电梯警报响起

规约至少应该包含下述属性：

- 如果第三层的按钮被按下，电梯将会到达该层。
- 电梯不可能同时在第一层和第二层。
- 如果电梯没有移动，门就会打开。
- 如果没有按钮被按下并且电梯在第四层，电梯将在该层等待直到按钮被按下。
- 无论何时电梯在楼层间被卡住，警报声将会响起直到电梯恢复移动。

练习 5.10.3 描述 4.15 节的通信协议要求的属性，特别是下面的属性：

- 如果前一个消息没有被接收者接收，发送者不会发送下一条消息。
- 如果由发送者发送的每条消息仅会有限多次丢失，那么每条消息最终会到达接收者。

5.11 扩展阅读

以下是一本软件规约方面的书：

V. S. Alagar, K. Periyasamy, *Specification of Software Systems*, Springer-Verlag, 1998.

Wolfgang Thomas 的两篇关于 Büchi 自动机和一元二阶逻辑的综述如下：

W. Thomas, Automata on infinite objects, in *Handbook of Theoretical Computer Science*, vol. B, J. van Leeuwen (ed)., Elsevier, (1990) 133–191.

W. Thomas, Languages, Automata, and Logic, in *Handbook of Formal Language Theory*, G. Rozenberg, A. Salomaa, (eds.), Volume 3, Springer-Verlag, 389–455.

规约的时序描述方法在下面的书中描述得很详细：

Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1991.

F. Kröger, *Temporal Logic of Programs*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

在下面的综述论文中能找到更为广义的关于时序逻辑的观点：

E. A. Emerson, Temporal and Modal Logic, in *Handbook of Theoretical Computer Science*, vol. B, J. van Leeuwen (ed.), Elsevier, (1990), 995–1072.

第6章
Software Reliability Methods

自动验证

她走啊走啊，走了很远，但是每当有岔道时，那两个路标肯定指着同一个方向，其中一个写着“通往特威度丹姆的寓所”，另一个写着“通往特威度迪的寓所”。

“我可以肯定，”爱丽丝恍然大悟，“他们住在同一栋房子里！”

刘易斯·卡洛尔《爱丽丝镜中奇遇记》

毫无疑问，软件系统的完全自动验证是一个非常理想的目标。我们希望拥有一种工具：它可以接受待验证系统及其规约作为输入，在没有人工干预的情况下，自动检查给定软件系统是否满足它的规约。但是根据可计算性理论（见第2章）可知，我们无法构造一个工具来广泛检验不同类别的程序。然而，理论限制并不能阻止我们寻找检验软件正确性的实际应用解决方案。

帮助我们实现这一目标的一些可能的思路包括：

- 限制验证的范围，关注自动验证算法可处理的较小类别的程序。在本章稍后讨论的模型检验中，这类程序一般只包含有限多个状态。
- 不再验证整个系统，而是关注其中某些简单但关键的部分。例如，可以仅验证：
  —基本核心算法；
  —系统的受限版本，如对变量取值、消息队列大小等设置（小）范围；
  —并发系统的底层通信协议。
- 使用抽象方法隐藏或删减程序中的一些细节信息，从而得到一个较为简单的程序（例如，一个有限状态系统）。这个抽象过程可以由人工完成，因为如果存在一个抽象算法总是能将验证问题约简为可判定问题，这会将与原先不可判定的结果相矛盾。这会导致实际系统和验证版本之间存在潜在的差别。因此，这样的验证过程可以提高，而不是绝对保证，对于被验证系统正确性的信心。
- 使用人工辅助的验证方法。例如，使用人工演绎证明以验证部分系统，如验证一个给定的抽象转换，并将验证问题约简为一个简单的可判断问题。我们将在10.1节深入讨论这个方法。

本章介绍了一组自动验证有限状态系统中属性的技术。Bochmann [16] 是最先提出将有限状态机（自动机）用于形式化方法的人之一，他使用有限状态机对通信协议进行建模。Zafiropulo [150] 给出了基于状态空间搜索的算法，用于分析有限状态机建模的通信协议。West 和 Zafiropulo [146] 使用这个算法验证了 CCITT X.21 通信协议。模型检验（model checking），即针对时序逻辑规约的程序算法验证，最先由美国的 Clarke 和 Emerson [28, 42] 以及法国的 Quielle 和 Sifakis [120] 分别独立提出。

即使我们将关注点局限于有限状态系统，在验证过程中往往还需要处理内在的高复杂度。一个有限状态的程序仍然可能包含巨大的状态数。并发系统特别复杂，运行中可能产生的状态数等于各个组件中状态数的乘积。因此，模型检验工具中经常使用各类启发式方法以试图解决状态空间爆炸问题。

6.1 状态空间搜索

考虑一个初始状态集合为 I 的有限状态系统。以下算法搜索状态空间，访问由初始状态可达的所有状态。其中使用 *new* 和 *old* 两个状态链表。

状态空间搜索算法

```
1  Let new contain the set of intial states I and old be empty;
2  While new is not empty do
3    Choose some state s from new;
4    Remove s from new;
5    Add s to old;
6    For each transition t that is enabled at s do
7      Apply transition t s, obataining s';
8      If s' is not already in new or in old then
9        Add s' to new;
```

在算法的第 3 行和第 9 行中，并未指明如何从 *new* 中选取一个状态 s，或是如何将新状态存入 *new*。这里有几种不同的实现方法。一种方法是将 *new* 作为队列实现，根据先进先出的顺序删除元素。同时，每个新元素添加到 *new* 队列的尾端。这种搜索策略一般称为广度优先搜索（BFS）。当要把状态加入 *new* 时采用以下方法：首先队列 *new* 中要包含初始状态，然后将与 I 中某个状态的距离（最短路径长度）为 1 的状态加到 *new* 队列的尾端，再然后将距离为 2 的状态加入，以此类推。

对图 2.1 使用 BFS 方法进行遍历搜索，初始状态 $I=\{r_1\}$。下表中的列 *new*、s 和 *old* 分别表示算法第 4 行在连续迭代过程中的变量值，最后一行表示算法执行结束时的变量值。

new	s	*old*
$\langle r_1 \rangle$	r_1	$\langle\rangle$
$\langle r_2 \rangle$	r_2	$\langle r_1 \rangle$
$\langle r_3, r_5 \rangle$	r_3	$\langle r_1, r_2 \rangle$
$\langle r_5, r_4 \rangle$	r_5	$\langle r_1, r_2, r_3 \rangle$
$\langle r_4, r_9, r_6 \rangle$	r_4	$\langle r_1, r_2, r_3, r_5 \rangle$
$\langle r_9, r_6, r_8 \rangle$	r_9	$\langle r_1, r_2, r_3, r_5, r_4 \rangle$
$\langle r_6, r_8, r_7 \rangle$	r_6	$\langle r_1, r_2, r_3, r_5, r_4, r_9 \rangle$
$\langle r_8, r_7 \rangle$	r_8	$\langle r_1, r_2, r_3, r_5, r_4, r_9, r_6 \rangle$
$\langle r_7 \rangle$	r_7	$\langle r_1, r_2, r_3, r_5, r_4, r_9, r_6, r_8 \rangle$
$\langle\rangle$		$\langle r_1, r_2, r_3, r_5, r_4, r_9, r_6, r_8, r_7 \rangle$

我们依次扩展状态（根据算法第 4 行，每次从链表 *new* 中选出的状态）的顺序为：r_1，r_2，r_3，r_5，r_4，r_9，r_6，r_8，r_7。事实上，这并不是 BFS 算法下唯一可能的排列顺序，因为某个状态的不同后继（如 r_2 的后继为 r_3 和 r_5）可能根据不同的添加顺序被加入到 *new* 中。

另一个策略是使用栈实现 *new*，即删除元素时基于后进先出的顺序。这种搜索策略称为深度优先搜索（DFS）。再次考虑图 2.1，从 $I=\{r_1\}$ 开始使用 DFS 搜索图空间。下表给出了算法第 4 行每个迭代过程中 *new*、s 和 *old* 的变量值，以及最终的执行结果。

new	s	*old*
$\langle r_1 \rangle$	r_1	$\langle\rangle$

$\langle r_2\rangle$ r_2 $\langle r_1\rangle$
$\langle r_3, r_5\rangle$ r_3 $\langle r_1, r_2\rangle$
$\langle r_4, r_5\rangle$ r_4 $\langle r_1, r_2, r_3\rangle$
$\langle r_8, r_5\rangle$ r_8 $\langle r_1, r_2, r_3, r_4\rangle$
$\langle r_5\rangle$ r_5 $\langle r_1, r_2, r_3, r_4, r_8\rangle$
$\langle r_9, r_6\rangle$ r_9 $\langle r_1, r_2, r_3, r_4, r_8, r_5\rangle$
$\langle r_7, r_6\rangle$ r_7 $\langle r_1, r_2, r_3, r_4, r_8, r_5, r_9\rangle$
$\langle r_6\rangle$ r_6 $\langle r_1, r_2, r_3, r_4, r_8, r_5, r_9, r_7\rangle$
$\langle\rangle$ $\langle r_1, r_2, r_3, r_4, r_8, r_5, r_9, r_7, r_6\rangle$

因此，我们依照 r_1，r_2，r_3，r_5，r_4，r_8，r_9，r_6，r_7 的顺序扩展状态。同样，这并不是使用DFS扩展状态唯一可能生成的顺序。

如上算法中所示的简单搜索方法可以用于检验有限状态系统的不变量。这些系统属性可以用形如$\Box\varphi$的LTL公式表示，其中φ是一阶或命题公式。接下来，需要检验先前算法中初始状态（第1行）和每一个加入集合 *new* 的新状态（第9行）是否满足φ。

假设搜索算法找到一个从初始状态出发可达但不满足φ的状态。我们不仅关注这个状态是否存在，还关注这个状态是如何生成的。换言之，生成一条从初始状态集合 I 中某个状态出发并且最终到达一个违背φ的状态的路径是非常有价值的。这样的路径对应于某个执行的前缀。它除了能找出是否存在错误，还能发现错误发生的原因。我们可以通过为每个状态记录一条边来重现这条路径，这条边回溯地指向在搜索过程中首次发现这个状态的唯一的先驱节点[⊖]。也就是说，在算法第9行中，我们保留一个从 s' 到 s 的回溯指针（需要注意的是这里的指针方向与状态空间图中边的方向相反）。图6.1中给出了DFS算法生成的节点序列和相应的回溯边。例如，假设状态 r_9 不满足φ，可以回溯得到从初始状态（在本例中仅有一个）开始的路径 r_1，r_2，r_5，r_9。

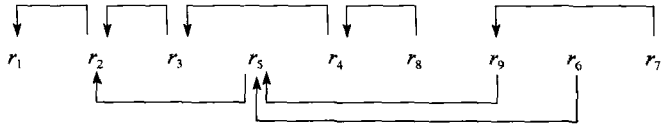


图 6.1 包含回溯边的节点的DFS遍历

BFS算法能生成路径长度最短的反例，而DFS不能保证这个属性。另外，检验属性比检验不变量更加复杂，这将在下面几节中讨论。

6.2 状态表示方法

在本章介绍的自动验证算法中，如何表示状态极为重要。我们需要用足够的信息表示每个状态，以便根据其可能发生的转换计算它的后继状态。同样，区分一个状态与其他不同的状态也很重要。如果不能判断我们是否再一次到达了同一个状态，搜索算法将无法终止。

一个典型的状态表示方法包含对程序变量的赋值、程序计数器、系统中的消息传递、消息队列。使用栈或树这类复杂数据结构的程序常常导致生成的状态空间过大，而无法自动进行分析。

一个微妙的而常被忽略的注意点是，不应该使用一种区分本质相同的状态的表示方法。为了描述这个问题，考虑以下示例[67]：拥有两个通信队列 *qu*0 和 *qu*1 的系统。系统使用含有两条队列信息的一个列表进行建模，而不是使用两个不同的队列变量。列表中的消息存储顺序由它们的到达顺序而定。每条消息标记为0或1，因此，对单条队列内容进行线性搜索就可以检验

⊖ 通常在模型检验中，递归实现DFS时反例存留在递归栈中。

或重新获取每一个原始队列的头。

这种看似简单的表示方法将产生严重的后果。考虑 $qu0$ 和 $qu1$ 各自接受 n 条消息的情况。这可以由一个用两个单独的队列变量描述相关队列内容的状态表示。由于不同队列中消息的到达顺序不同，在单列表表示方法中，两个队列的消息有很多种不同的交错方式。在这里，这种顺序是无关的（如果有影响的话，它可以从状态序列中推导而得）。与分别用一个状态表示不同的队列不同，一个单列表表示方法可以有 $(2n)!/(n!^2)$ 个不同的状态。例如，当使用单列表表示两条队列的不同交错顺序时，如果 $n=5$，将生成 252 个状态。

状态在存储器中的存储方式对模型检验的效率有着深远影响。为了快速地重新检索状态，同时保持内存大小的可控性，通常使用哈希表存储状态。同样，压缩技术也被应用于减少所使用的状态空间 [65]。

6.3 自动机结构体系

这里介绍的模型检验方法基于自动机理论。它允许使用同样的表示方法描述待验证系统和规约，也就是 Büchi 自动机。自动机理论与其他研究领域间的联系，以及大量的相关研究成果，使得我们可以开发新的模型检验算法。其他的相关成果将自动机理论与多种逻辑 [141] 联系在一起，如时序逻辑、一元一阶逻辑、一元二阶逻辑和正则表达式。这有助于在模型检验算法中使用这些形式化方法表示规约。

Kurshan [4] 以及 Vardi 和 Wolper [144] 分别独立提出了模型检验的自动机理论体系。此外，本章中的相关介绍也参考了 Alpern 和 Schneider [5] 的研究成果。我们将介绍一些自动机理论的基本知识，并说明在这个体系下如何完成模型检验。我们将特别介绍用自动机描述线性时序逻辑的模型检验。

回顾一下，我们可以使用基于相同字母表的自动机表示状态空间和系统规约。如果系统 $\mathcal{A}$ 的语言包含于规约 $\mathcal{B}$ 的语言，即

$$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B}) \tag{6.1}$$

那么系统模型 $\mathcal{A}$ 满足规约 $\mathcal{B}$。

令 $\overline{\mathcal{L}(\mathcal{B})}$ 为语言集 $\Sigma^{\omega} \setminus \mathcal{L}(\mathcal{B})$ 所有不被 $\mathcal{B}$ 接受的字的集合，即语言 $\mathcal{L}(\mathcal{B})$ 的补集。式 (6.1) 中的包含关系可以改写为：

$$\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{B})} = \varnothing \tag{6.2}$$

这意味着不存在 $\mathcal{A}$ 可以接受而不被 $\mathcal{B}$ 允许的字。如果交集不为空，那么其中的任何元素都是式 (6.1) 的反例。

实现式 (6.2) 中的语言交集比实现式 (6.1) 中的语言包含更简单；检验两个自动机相交的语言为空的算法比检验语言包含关系的算法要简单。当然，在 6.5 节中可知，很难构建 Büchi 自动机的补集。然而，当规约的来源是一个 LTL 公式 φ 时，我们可以避免互补计算。这可以通过对被检验公式 φ 进行取反而完成，即直接将 $\neg\varphi$ 转换到自动机 $\overline{\mathcal{B}}$，而不是将 φ 转换到自动机 $\mathcal{B}$ 再计算补集。

对表示原始待验证系统的模型 $\mathcal{A}$ 建模时，可能引入建模错误，因此我们需要仔细评估自动验证过程的结果。当 $\mathcal{A}$ 不完全规约建模系统时，可能出现问题，会导致一些执行不被 $\mathcal{L}(\mathcal{A})$ 包含。这可能允许包含关系式 (6.1) 被满足，模型检验过程报告模型满足规约，即使实际系统并非如此。

相反，$\mathcal{A}$ 也可能过度规约建模系统。结果，当发现反例时，我们需要检查建模错误并未引入一个错误的反例，这称为误报 (false negative)。当 $\mathcal{L}(\mathcal{A})$ 包含一个不符合建模系统行为的执行时，就有可能会发生这种情况。将反例与建模系统作比较相对简单。因此，在模型检验中找出反

例比获得程序模型满足规约的结论更具有价值。

在下一节中将讨论 Büchi 自动机在交、并和补操作下闭包。这意味着，存在一个自动机可以完全接受两个给定自动机的语言的交集或并集，存在一个自动机可以完全接受给定自动机的语言的补集。有时，我们非正式地称对两个自动机进行交或者并操作，这意味着我们构建了一个自动机，它可以接受两个自动机的语言的交集或并集。

式（6.2）中的正确性标准引入了如下的模型检验一般策略：

> 首先，计算自动机$\mathcal{B}$的补集，即构建一个可以接受语言$\overline{\mathcal{L}(\mathcal{B})}$的自动机$\overline{\mathcal{B}}$。（或者，直接生成规约$\mathcal{B}$的补集。）然后，计算自动机$\mathcal{A}$与$\overline{\mathcal{B}}$的交集。如果交集为空，则规约$\mathcal{B}$适用于$\mathcal{A}$。否则，将非空交集的可接受字作为一个反例。

下一节中将对如何基于 Büchi 自动机实现这一策略进行详细说明。

6.4 合并 Büchi 自动机

规约机制的一个重要性质是对布尔操作“与”、“或”、“非”闭包。在语言中，这些操作分别对应于交、并和补。以交集为例，令 φ、ψ 为序列上的规约，使得$\mathcal{L}(\varphi)$ 是满足属性 φ 的序列，记为 $\{\sigma \mid \sigma \models \varphi\}$，有

$$\mathcal{L}(\varphi \wedge \psi) = \{\sigma \mid \sigma \models \varphi \text{ and } \sigma \models \psi\} = \mathcal{L}(\varphi) \cap \mathcal{L}(\psi)$$

我们将会看到 Büchi 自动机在这些操作下完全闭包。这一点非常重要，因为从式（6.2）中我们可知模型检验可以被转换为自动机的交集和补集。

基于共同的字母表 Σ，给定一对自动机$\mathcal{A}_1 = \langle \Sigma, S_1, \Delta_1, I_1, L_1, F_1 \rangle$ 和$\mathcal{A}_2 = \langle \Sigma, S_2, \Delta_2, I_2, L_2, F_2 \rangle$。如果 $S_1 \cap S_2 = \varnothing$，那么可以构造并集自动机$\mathcal{A}_1 \cup \mathcal{A}_2$。（如果 $S_1 \cap S_2 \neq \varnothing$，则可以对状态重命名。）$\mathcal{A}_1 \cup \mathcal{A}_2$ 的语言是各个自动机的语言的并集，即$\mathcal{L}(\mathcal{A}_1 \cup \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$（因此$\mathcal{A}_1 \cup \mathcal{A}_2$ 又被称为并集自动机）。

并集自动机$\mathcal{A}_1 \cup \mathcal{A}_2$ 定义为多元组：

$$\langle \Sigma, S_1 \cup S_2, \Delta_1 \cup \Delta_2, I_1 \cup I_2, L_1 \cup L_2, F_1 \cup F_2 \rangle$$

其中，标记 $L = L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ 与 L_1 和 L_2 一致，也就是说，如果 $s \in S_1$，那么 $L(s) = L_1(s)$，如果 $s \in S_2$，那么 $L(s) = L_2(s)$。并集自动机是根据$\mathcal{A}_1$ 行动还是根据$\mathcal{A}_2$ 行动，这是个非确定性选择。从 I_1 或 I_2 中选择一个初始状态，之后的行动依据适当的自动机作出相应的选择。当自动机$\mathcal{A}_1$ 和$\mathcal{A}_2$ 表示规约时，它们的并集对应于规约的析取（disjunction）：或者根据规约$\mathcal{A}_1$ 行动，或者根据规约$\mathcal{A}_2$ 行动。此并集包含了两个自动机的语言，因此允许系统根据两者间任意一个规约行动。n 个自动机的并集定义可按照上述方法扩展而来。

接受交集$\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ 的交集自动机$\mathcal{A}_1 \cap \mathcal{A}_2$，对应于$\mathcal{A}_1$ 和$\mathcal{A}_2$ 给定规约的合取（conjunction）。交集只包含同时被两个自动机接受的字。因此，交集自动机的每个运行需要模拟统一输入下两个同时发生的运行，自动机$\mathcal{A}_1$ 上的运行和$\mathcal{A}_2$ 上的运行。此刻，我们将忽略接受条件，并在稍后讨论。交集自动机的每个状态都与$\mathcal{A}_1$ 中的一个状态和$\mathcal{A}_2$ 中的一个状态相关。在两个自动机同时发生的一个运行中，输入序列必须满足每一个状态的（标记在状态上的）状态条件。因此，交集中状态上的条件是其各个组件的条件的合取。

考虑如图 6.2 所示的两个自动机。状态间的四种组合如下：

$$L(\langle q_0, q_2 \rangle) = L_1(q_0) \wedge L_2(q_2) = A \wedge (\neg B) \wedge (\neg A) \wedge B \quad = false$$

$$L(\langle q_0, q_3 \rangle) = L_1(q_0) \wedge L_2(q_3) = A \wedge (\neg B) \wedge (A \vee (\neg B)) \quad = A \wedge (\neg B)$$

$$L(\langle q_1, q_2 \rangle) = L_1(q_1) \wedge L_2(q_2) = (\neg A) \wedge (\neg A) \wedge B \quad = (\neg A) \wedge B$$

$$L(\langle q_1, q_3 \rangle) = L_1(q_1) \wedge L_2(q_3) = (\neg A) \wedge (A \vee (\neg B)) \quad = (\neg A) \wedge (\neg B)$$

图 6.2 左侧的自动机要求其中的每个状态或者满足 $A \wedge (\neg B)$ 或者满足 $\neg A$。另外，对于无限多个状态，满足 $A \wedge (\neg B)$。右侧自动机要求初始状态和每个偶数状态（从 0 开始计数）满足 $(\neg A) \wedge B$，其余的状态必须满足 $A \vee (\neg B)$。

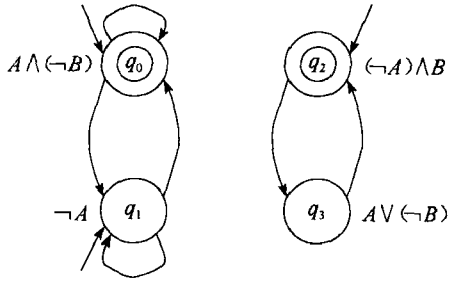


图 6.2 两个自动机的交集

需要注意的是对于状态组 $\langle q_0, q_2 \rangle$，其条件为 *false*。所以，没有任何运行可以通过这个状态组，因为不存在一个赋值可以满足公式 *false*。我们可以安全移除标记为 *false* 的状态，以及它们的入边和出边。当 $\mathcal{A}_1$ 中有 q 到 q' 的边且 $\mathcal{A}_2$ 中有 r 到 r' 的边时，交集中两个状态 $\langle q, r \rangle$ 和 $\langle q', r' \rangle$ 之间存在边。交集中的初始状态组的各分量分别是 $\mathcal{A}_1$ 和 $\mathcal{A}_2$ 中的初始状态。如图 6.3 所示的是最终得到的不包含接受状态的自动机。

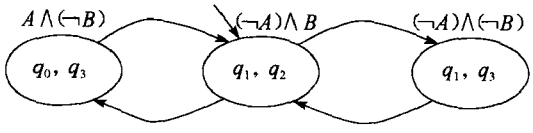


图 6.3 交集的中间状态

此刻，我们仍未讨论如何判断哪些状态可以被接受。首先我们将介绍两种错误尝试来说明这个微妙的问题。在第一种尝试中，我们设定当 $\mathcal{A}_1$ 分量或 $\mathcal{A}_2$ 分量被接受时，交集中的状态被接受。这将允许自动机接受一个无限多次到达自动机 $\mathcal{A}_1$ 或 $\mathcal{A}_2$ 的接受状态的字。这样，q_0 使得状态 $\langle q_0, q_3 \rangle$ 成为接受状态，q_2 使得状态 $\langle q_1, q_2 \rangle$ 成为接受状态。在 Büchi 自动机的接受条件中，这将允许接受至少通过二者之一的一个无限序列，如一个仅循环通过 $\langle q_1, q_2 \rangle$ 和 $\langle q_1, q_3 \rangle$ 的序列。这样的序列不包含满足 $A \wedge (\neg B)$ 的状态，而图 6.2 中左侧自动机要求这样的状态出现无限多次。

在第二种尝试中，我们设定交集的接受状态的各个分量分别被 $\mathcal{A}_1$ 和 $\mathcal{A}_2$ 接受。这种情况的问题是两个自动机 $\mathcal{A}_1$ 和 $\mathcal{A}_2$ 可能永远无法同时到达它们的接受状态。在图 6.3 中，状态组 $\langle q_0, q_3 \rangle$ 包含 $\mathcal{A}_1$ 的接受状态 q_0，而 q_3 不被 $\mathcal{A}_2$ 接受。同样，$\langle q_1, q_2 \rangle$ 包含 $\mathcal{A}_2$ 的接受状态 q_2，而 q_1 不被 $\mathcal{A}_1$ 接受。第三组 $\langle q_1, q_3 \rangle$ 中的状态 $\mathcal{A}_1$ 和 $\mathcal{A}_2$ 都不接受。因而，在此例中，没有状态有资格被接受。

因此，我们需要一些方法以保证可以无限多次访问 F_1 和 F_2 接受集中的状态，尽管不一定要求同时发生。我们给出的解决方法由两步组成：首先我们定义一个 Büchi 自动机的广义版本，它允许我们处理多重接受条件。然后，我们将说明如何将在扩展条件中定义的自动机转换为简单 Büchi 自动机。这里介绍的广义自动机的定义同样可以用于后文中介绍的从 LTL 到自动机的转换算法。

6.4.1 广义 Büchi 自动机

不同于 Büchi 自动机只有一个接受集，广义 Büchi 自动机构允许有多个接受集合。它的结构是六元组 $\langle \Sigma, S, \Delta, I, L, F \rangle$，其中 $F = \{f_1, f_2, \cdots, f_m\}$，每个 f_i 都是 S 的一个子集 $(1 \leqslant i \leqslant m)$，即 $f_i \subseteq S$。其他分量和简单 Büchi 自动机中的定义相同。一个可接受的运行必须无限多次地通过每一个 F 中的集合。相应的形式化描述如下，如果对于任意 $f_i \subseteq F$，有 $inf(\rho) \cap f_i \neq \varnothing$，那么广义 Büchi 自动机中的运行 ρ 是可被接受的。

显然，Büchi 自动机是广义 Büchi 自动机的特殊情况。不过，稍后我们将说明对广义 Büchi 自动机的接受条件的定义并没有增强普通 Büchi 自动机的表达能力。我们还会给出从广义 Büchi 自动机到接受相同语言的简单 Büchi 自动机的转换。

在为两个 Büchi 自动机的交集而构造的广义 Büchi 自动机中，我们令接受集合 f_1 表示第一个分量属于 F_1 的状态，即 $F_1 \times S_2$，令接受集合 f_2 表示第二个分量属于 F_2 的状态，即 $S_1 \times F_2$。在图 6.3 中，第一个接受集合只有一个状态 $\{\langle q_0, q_3 \rangle\}$。（需要注意的是，状态 $\langle q_0, q_2 \rangle$ 由于

false 条件而被移除。）第二个接受集合是 $\{\langle q_1, q_2\rangle\}$。

6.4.2 将广义 Büchi 自动机转换为简单 Büchi 自动机

这里我们介绍一种简单的转换方法 [34]，将一个广义 Büchi 自动机 $\langle \Sigma, S, \Delta, I, L, F\rangle$ 转换为一个（简单）Büchi 自动机。如果接受集合的数目 $|F|$ 是 m，我们构建 m 个状态集合 S 的独立副本，即 $\bigcup_{i=1,m} S_i$，其中 $S_i = S\times\{i\}(1\leqslant i\leqslant m)$。因此，$S_i$ 中的状态记为 (s, i)。形如 $\oplus_m$ 的加操作变为 $i\oplus_m 1 = i+1$，其中 $1\leqslant i\leqslant m$，而 $m\oplus_m 1 = 1$。这个运算符允许我们在 1 到 m 间循环计数。在构建的 Büchi 自动机的一个运行中，当访问 S_i 中的状态时，如果出现 f_i 中状态的副本，我们跳至 $S_{i\oplus_m 1}$ 中相应的后继状态。否则，我们跳至 S_i 中相应的后继状态。因此，以增序访问 F 中所有集合的接受状态，使得自动机循环经过 m 个备份。

我们需要选出无限多次经过从 S_1 到 S_m 每一个副本的接受状态。由于从一个集合到下一个集合的跳转与某个 f_i 中接受状态的出现相一致，这保证了所有接受集合都将出现无限多次。我们不能选择 S_i 集合中的一个元素作为 Büchi 接受集合，因为这样可能会接受一个永久停留在 S_i 而从不出现 f_i 中任何状态的运行。但我们可以对任意 $1\leqslant i\leqslant m$ 选择笛卡儿乘积 $f_i\times\{i\}$。这保证了我们经过 $f_i\times\{i\}$ 中的一个状态并可到达 $S_{i\oplus_m 1}$ 中的一个状态。为了重新访问 $f_i\times\{i\}$ 中的状态，我们需要再一次循环遍历所有其他的备份。当广义 Büchi 自动机的接受集合 F 中的集合为空时，我们定义转换结果为 $\langle \Sigma, S, \Delta, I, L, S\rangle$，即生成的 Büchi 自动机的所有状态都是接受状态。

我们对不同集合中接受状态的出现附加了先后次序，这可能初看来很奇怪。这并不意味着 F 中不同集合的接受状态按照这样的顺序出现在输入序列中。它只说明我们按这样的顺序考虑它们。例如，如果有状态 $(s, 2)\in S_2$，其中 $s\in f_2$，我们跳转至某一状态 $(r, 3)\in S_3$。然后，如果有状态 $(q, 3)\in S_3$ 出现，其中 $q\in f_2$ 或 $q\in f_1$，我们将继续停留在 S_3。需要注意的是，如果有无限多个状态其第一个组件属于各个接受集合 f_i，我们将不去利用所有这些状态以在不同备份间跳转，这并不会带来问题。如果在某一时刻后，没有更多属于 f_i 的状态，那么自然也就不会接受相应的输入。

由广义 Büchi 自动机 $\langle \Sigma, S, \Delta, I, L, \{f_1, \cdots, f_m\}\rangle$ 转换而得的包含非空接受条件集合的 Büchi 自动机为 $\langle \Sigma, S\times\{1, \cdots, m\}, \Delta', I\times\{1\}, L', f_1\times\{1\}\rangle$（$m=0$ 的情况在上文中已给出），其中 $L'(q, i) = L(q)$，且如果 $(q, q')\in\Delta$ 并满足下列条件之一：

1. $q\in f_i$ 且 $j = i\oplus_m 1$
2. $q\notin f_i$ 且 $j = i$

则有 $(\langle q, i\rangle, \langle q', j\rangle)\in\Delta'$。

现在，我们可以定义两个 Büchi 自动机的交集为一个广义 Büchi 自动机。对应的交集构建为：

$$\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, S_1\times S_2, \Delta, I_1\times I_2, L, \{F_1\times S_2, S_1\times F_2\}\rangle$$

交集的转换关系 Δ 定义如下：

$$(\langle l, q\rangle, \langle l', q'\rangle)\in\Delta \text{ 当且仅当 } (l, l')\in\Delta_1, \text{且 } (q, q')\in\Delta_2$$

交集中各个状态 $\langle l, q\rangle$ 的标签 $L_1(l)\wedge L_2(q)$，记为 $L(l, q)$。标记为 *false* 的状态及其入边和出边被删除。将图 6.2 中的自动机进行交集运算，并将广义 Büchi 自动机转换为一个简单 Büchi 自动机后的结果如图 6.4 所示。

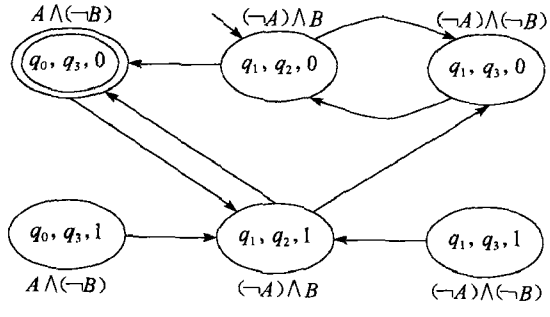


图 6.4 图 6.2 中自动机的交集

需要注意的是，自动机并集的大小与成员自动机大小的和成正比。自动机交集的大小与成员自动机大小的乘积成正比。

式（6.2）中的交操作通常对应于更严格的

情况，其中表示被建模系统的自动机中的所有状态都可以被接受。在这种受限情况中，自动机$\mathcal{A}_1$的所有状态是接受状态，而自动机$\mathcal{A}_2$的状态不受限制，则有：

$$\mathcal{A}_1 \cap \mathcal{A}_2 = \langle \Sigma, S_1 \times S_2, \Delta, I_1 \times I_2, L, S_1 \times F_2 \rangle \tag{6.3}$$

其中$(\langle l, q\rangle, \langle l', q'\rangle) \in \Delta$当且仅当$(l, l') \in \Delta_1$，且$(q, q') \in \Delta_2$，对于交集中的每个状态$(l, q)$，有$L(l, q) = L_1(l) \wedge L_2(q)$。显而易见，这是一个简单Büchi自动机。因此，其接受状态是第二个成员可被接受的状态组。然而，交操作更通用的情况适用于对附加了公平性约束的系统建模（见6.10节）。在这种情况下，并不需要自动机系统中所有状态都必须可被接受。

计算Büchi自动机补集的细节非常复杂。其主要原理将在下一节中介绍。更多的细节可参见[134]，而更有效的算法可参见[124]。补集自动机$\overline{\mathcal{B}}$的大小是$\mathcal{B}$大小的指数级。在一些实现中（如[65]），建议使用者直接给出$\overline{\mathcal{B}}$的自动机，而不是$\mathcal{B}$的自动机。这种情况下，用户对坏行为而不是好行为进行规约。在COSPAN[79]中使用的另一个方法是使用允许简单补集计算的ω自动机的扩展版本。

最后，自动机$\mathcal{B}$可以通过一些规约语言的转换而得到，如LTL或一元二阶逻辑。此时，不再先转换属性φ到$\mathcal{B}$再计算$\mathcal{B}$的补集，而是如式（6.2）中要求的对$\neg\varphi$进行转换，这将直接给出补集语言的自动机。一个将LTL转换到自动机的有效算法将在稍后介绍。

6.5 Büchi自动机求补

为了完备性，我们在这里介绍Büchi自动机求补操作的原理（但不给出全部细节）。如前一节中所说，由于可能的指数级爆炸，一些实现中避免了显式求补。因此，读者可以安心跳过以下数学细节和过程，直接进入下一节。

在字符串集合上的全等性（congruence）“$\sim$”是一个等价关系，满足$(x_1 \sim y_1 \wedge x_2 \sim y_2) \rightarrow x_1 . x_2 \sim y_1 . y_2$。给定Büchi自动机$\mathcal{A} = \langle \Sigma, S, \Delta, I, L, F\rangle$，令$u \in \Sigma^*$，$q, q' \in S$，包含由$u$中字母标记的从状态$q$到状态$q'$的边的路径，记为$q \overset{u}{\rightsquigarrow} q'$。经过$F$中接受状态的路径，记为$q \overset{u}{\rightsquigarrow}^F q'$。如果对每个$q, q' \in S$，$q \overset{u}{\rightsquigarrow} q'$仅当$q \overset{v}{\rightsquigarrow} q'$，且$q \overset{u}{\rightsquigarrow}^F q'$仅当$q \overset{v}{\rightsquigarrow}^F q'$，记为$u \cong v$。很容易验证“$\cong$”是一个等价关系。

现在证明“$\cong$”是全等的。我们将给出“$\overset{u}{\rightsquigarrow}$”的证明，同理可证“$\overset{u}{\rightsquigarrow}^F$”。令$u_1 \cong v_1$且$u_2 \cong v_2$，$q, q'$为任一状态组，我们现在需要证明存在标记为$u_1 . u_2$的从$q$到$q'$的路径，仅当存在标记为$v_1 . v_2$的从$q$到$q'$路径。由先前的等价性可知，对任一状态$r$，$q \overset{u_1}{\rightsquigarrow} r$当且仅当$q \overset{v_1}{\rightsquigarrow} r$，并且$r \overset{u_2}{\rightsquigarrow} q'$当且仅当$r \overset{v_2}{\rightsquigarrow} q'$。

从“$\cong$”关系的定义可知其等价类的数目有限（但可能是巨大的，比如$\mathcal{A}$中状态数的指数级）。“$\cong$”的每一个等价类U是有限个字的集合（或语言）。

对于“$\cong$”的每一组等价类U，V，要么$U . V^\omega \subseteq \mathcal{L}(\mathcal{A})$，要么$U . V^\omega \subseteq \overline{\mathcal{L}(\mathcal{A})}$。为了证明这点，令$u . v_1 . v_2 \cdots$和$u' . v_1' . v_2' \cdots$是$U . V^\omega$中两个无限字符串，其中$u, u' \in U$，$v_i, v_i' \in V$，$i \geqslant 1$。然后，由于$u \cong u'$和$v_i \cong v_i'$，其中$i \geqslant 1$，存在$\mathcal{A}$的运行$q_0 \overset{u}{\rightsquigarrow} q_{j_1} \overset{v_1}{\rightsquigarrow} q_{j_2} \overset{v_2}{\rightsquigarrow} \cdots$，仅当存在运行$q_0 \overset{u'}{\rightsquigarrow} q_{j_1} \overset{v_1'}{\rightsquigarrow} q_{j_2} \overset{v_2'}{\rightsquigarrow} \cdots$。此外，我们可以选择两个运行使得对每一个$i \geqslant 1$，有$q_{j_i} \overset{v_i}{\rightsquigarrow}^F q_{j_{i+1}}$仅当$q_{j_i} \overset{v_i'}{\rightsquigarrow}^F q_{j_{i+1}}$。因此，仅当第二个运行被接受时，第一个运行被接受。相应地，$u . v_1 . v_2 \cdots$在$\mathcal{L}(\mathcal{A})$中仅当$u' . v_1' . v_2' \cdots$在$\mathcal{L}(\mathcal{A})$中。所以，如果一个$U . V^\omega$中的字符串在$\mathcal{L}(\mathcal{A})$中，那么所有$U . V^\omega$中的字符串在$\mathcal{L}(\mathcal{A})$中。

令w是一个无限字。则存在“$\cong$”关系下的一组等价类U，V使得$w \in U . V^\omega$。为了证明这点，令$ix(x, y)$是“$\cong$”关系下包含了从第x个到第y个字母的子字$w[x..y]$的等价类。这

样，如果 $ix(x, y)=ix(x', y')$，那么 $w[x, y]\cong w[x', y']$。由于有有限多个等价类，ix 可以将无限多个配对映射到一个有限集合中。Ramsey 理论 [57] 证明把自然数组的无限集合分区到有限数目的集合中，存在自然数的一个无限序列 m_1，m_2，m_3，…，使得这个序列中的每一组数字，特别是每一相邻组，都在同一个分区集合中。这样，存在一个无限序列 m_1，m_2，m_3，…使得对任一 i，$j\geqslant 1$ 有 $ix(m_i, m_{i+1}-1)=ix(m_j, m_{j+1}-1)$。我们可以选择 $U=ix(0, m_1-1)$，$V=ix(m_1, m_2-1)$。

为了求 $\mathcal{A}$ 的补，我们可以构建（根据如上“$\cong$”的定义）形如 VU^{ω} 的语言的自动机并集，其中 U，V 是“$\cong$”的等价类，且 $VU^{\omega}\cap\mathcal{L}(\mathcal{A})=\varnothing$。

6.6 检验空集

在本书中，我们介绍了如何将模型检验转化为检验两个自动机的交集是否为空：其中一个自动机表示系统，另一个自动机表示规约的补集，见式 (6.2)。这样，我们需要一个实现空集检验的算法。此外，如果交集非空，我们希望得到反例。尽管 Büchi 自动机的一个执行可以是无限的，但我们希望得到可以有限表达的反例。

考虑 Büchi 自动机 $\mathcal{A}=\langle\Sigma, S, \Delta, I, L, F\rangle$。如 6.4 节中所示，我们可以假设 Büchi 自动机中没有一个状态的标记是等价于 *false* 的公式。在最普通的形式中，每个状态标记为命题变量 AP 上的任意布尔公式，这使得检验此公式是否等价于 *false* 成为复杂的问题。这个问题的时间复杂度是 NP 完备的，也就是说，至今尚未发现比指数级时间更好的解决方法。因此，我们唯一使用的状态标签是命题变量和否定命题变量的合取命题。特别是，6.8 节中自动 LTL 转换算法的结果符合这一形式。两个此类自动机交集的结果同样也符合这样的要求。在此条件下，是否等价于 *false* 的检验，可以在标记的长度的线性时间内完成。

令 ρ 是 $\mathcal{A}$ 的一个可接受运行，那么 ρ 包含了无限多个 F 中的状态。由于 S 是有限的，因此存在 ρ 的某个后缀 ρ'，使得 ρ' 的每个状态都出现无限多次。这意味着，ρ' 的每个状态都可以由 ρ' 中的另一个状态到达。因此，ρ' 中状态包含于 $\mathcal{A}$ 图的强连通组件中。该组件从初始状态出发可达，并且包含接受状态。反过来说，任何一个从初始状态可达并包含接受状态的非平凡强连通组件可以生成自动机的一个可接受运行。这样，检验 $\mathcal{L}(\mathcal{A})$ 的非空性等价于在图 $\mathcal{A}$ 中寻找一个从初始状态出发可达并包含接受状态的强连通组件。

这个发现的精妙之处在于，如果语言 $\mathcal{L}(\mathcal{A})$ 是非空的，那么就存在一个可以用有限方法表示的反例。这个反例是由一个有限前缀和一个周期性状态序列构成的运行。换句话说，反例是形如 $\sigma_1\sigma_2^{\omega}$ 的序列，其中 σ_1 和 σ_2 是有限序列。这样的序列称为*终极周期序列*。Tarjan 的深度优先搜索 (DFS) 算法 [139] 可以用于寻找强连通组件。这样，可以按照如下方法完成模型检验：

1. 构建表示被建模系统的自动机 $\mathcal{A}$。
2. 构建表示规约补集的自动机 $\overline{\mathcal{B}}$。
3. 构建交集自动机 $C=\mathcal{A}\cap\overline{\mathcal{B}}$。
4. 应用 Tarjan 的算法寻找从 C 中初始状态出发可达的强连通组件。
5. 如果未发现包含接受状态的强连通组件，则称模型 $\mathcal{A}$ 满足规约 $\mathcal{B}$。
6. 否则，选择 C 中可达的强连通组件 S。构建从 C 中某一初始状态出发到达 S 中某个接受状态 q 的路径 σ_1。构建从 q 到自身的循环。由于 S 是强连通组件，这样的循环必然存在。令 σ_2 为不包含第一个状态 q 的循环。然后，就可以称 $\sigma_1\sigma_2^{\omega}$ 是被 $\mathcal{A}$ 接受但不满足规约 $\mathcal{B}$ 的反例。

一个很重要的发现是，在开始计算交集和检验非空性之前，不需要完成系统构建或是自动机的相交。有时这称为“即时”(on-the-fly) 模型检验。当需要交集计算时，自动机系统中的新状态将按需构建。这样，有时可以检验因自动机模型过大而无法在内存中放下的某些系统。如果在交集完成前就已发现错误，那么将不再需要完成整个状态空间的构建。而且，由于属性相关自动机决定了乘积的构建，因此自动机系统的某些部分可以简单地不纳入交集运算。相关算法的更多细节可参见 [34]。

6.7 模型检验范例

考虑图 6.5 中左侧的交通信号灯系统模型，它不同于 5.4 节中的交通灯系统。这里，*yellow* 灯开始亮时，*green* 灯依旧未灭。命题变量为 *re*、*ye* 和 *gr*，分别对应于 *red*、*yellow* 和 *green*。自动机系统的所有状态都可被接受。也就是说，任何一个无限序列都是系统的一个执行。事实上，本例中只有一个执行。因此，我们并没有对图 6.5 中模型自动机的接受状态进行标记。

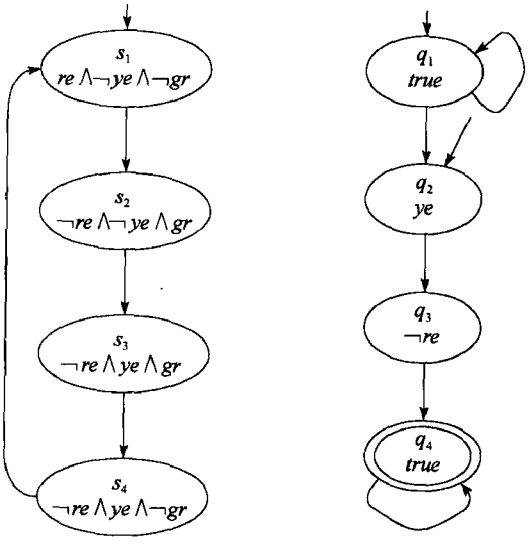


图 6.5 一个交通信号灯模型及其规约的否定

规约中我们希望检验断言：自动机总是从 *yellow* 到 *red*。相应地，我们生成 LTL 属性 $\square(ye \rightarrow \bigcirc re)$来检验交通信号灯系统。为了检验这个属性，我们检验模型中是否存在一个序列满足这个属性的否定形式，即 $\neg \square(ye \rightarrow \bigcirc re) = \Diamond(ye \wedge \bigcirc \neg re)$。图 6.5 右侧是表示这个属性的自动机。稍后，我们将介绍转换 LTL 属性到 Büchi 自动机的算法。

图 6.6 是两个自动机的交集，其中标签等价于 *false* 的节点被划去，它们的边也未画出。模型的初始状态是 $\langle s_1, q_1 \rangle$，其两个组件都是交集自动机的初始状态。($\langle s_1, q_2 \rangle$ 包含两个初始组件，但因 $L(s_1) \wedge L(q_2) = false$ 而被划去。) 同样还有些没有入边的非初始状态，如($\langle s_2, q_3 \rangle$，$\langle s_3, q_3 \rangle$)，以及没有出边的状态 ($\langle s_4, q_2 \rangle$)。由于它们不能出现在任何一个可接受的运行中，因此这些状态都是冗余的。本例中，我们处理的是简单（且常见）的情况，其中一个参与交集计算自动机（系统）的所有状态都被接受。因此，根据自动机属性，交集的接受状态是那些第二个组件被接受的状态。在本例中，是含有第二个组件 q_4 的状态。这里有两个非平凡强连通组件：

$$\{\langle s_1, q_1 \rangle, \langle s_2, q_1 \rangle, \langle s_3, q_1 \rangle, \langle s_4, q_1 \rangle\}$$

和

$$\{\langle s_1, q_4 \rangle, \langle s_2, q_4 \rangle, \langle s_3, q_4 \rangle, \langle s_4, q_4 \rangle\}$$

这两个组件都从初始状态可达（事实上，前一个组件中包含了初始状态）。前一个强连通组件中不包含接受状态，而后一个组件的所有状态都可以被接受。在此基础上，反例可以由有限前缀

$$\langle s_1, q_1 \rangle, \langle s_2, q_1 \rangle, \langle s_3, q_2 \rangle, \langle s_4, q_3 \rangle, \langle s_1, q_4 \rangle$$

与周期序列

$$\langle s_2, q_4 \rangle, \langle s_3, q_4 \rangle, \langle s_4, q_4 \rangle, \langle s_1, q_4 \rangle$$

组成。

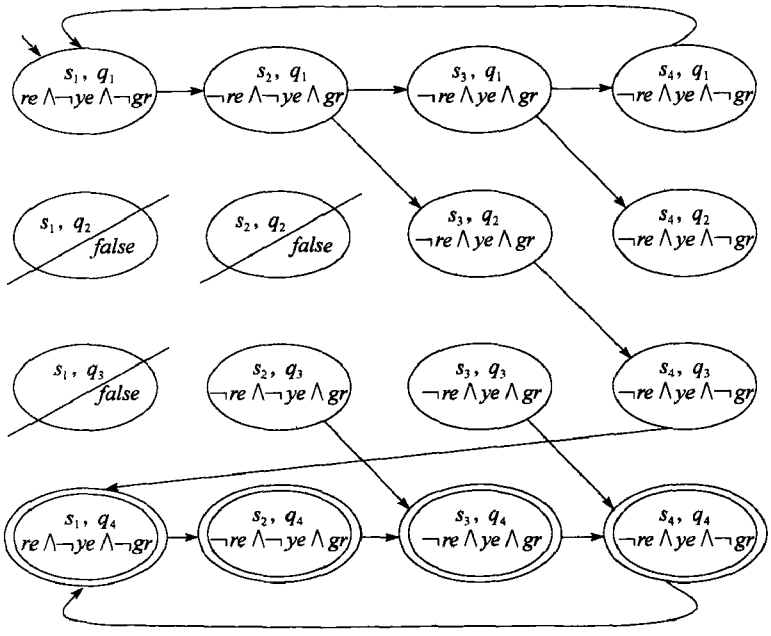


图 6.6 图 6.5 中自动机的交集

分析反例可知错误部分出现在前缀中，当 *yellow* 和 *green* 都亮时，从状态 $\langle s_3, q_2 \rangle$ 跳转至下一状态 $\langle s_4, q_3 \rangle$，这时 *yellow* 仍然亮着，而 *red* 未亮。

在把这个运行返回给工程师并向其抱怨出现错误之前，我们需要判断这是不是真的表明在交通信号灯系统中存在某个错误。仔细检查反例可知在一个排他的 *yellow* 状态后出现先前的 *red* 状态。在先前 *red* 状态后的第一个 *yellow* 状态中，*green* 灯仍然亮着。这里根据之前的规约，下一状态中我们需要 *red* 亮起。然而在下一状态中，*green* 灯灭掉，而 *yellow* 灯仍然亮着。只有在子序列状态中，*yellow* 跳转到 *red*。看起来似乎所使用的 LTL 属性并未描述出预定的规约，而反例的行为并未暴露出任何问题。在此例中，应该重新描述 LTL 属性（或许表示为 $\square(ye \rightarrow ye \cup re)$）。

6.8 将 LTL 转换为自动机

现在，我们介绍将（命题）LTL 规约转换为广义 Büchi 自动机的算法 [52]。令 φ 为要转换为广义 Büchi 自动机的 LTL 规约。我们首先来直观地描述这个转换算法。

对生成的自动机中每一个节点 s，我们附加一个公式 $\eta(s)$。给定某个序列 ξ 上的一个可接受运行，如果这个运行到达状态 s 且有后缀 ξ^i，那么有 $\xi^i \models \eta(s)$。公式 $\eta(s)$ 形如

$$\left(\bigwedge_{i=1..m} \nu_i\right) \wedge \bigcirc\left(\bigwedge_{j=1..n} \kappa_j\right)$$

这有助于计算满足 $\bigwedge_{j=1..n} \kappa_j$ 的 s 的后继节点。需要注意的是当 $m=1$ 且 $n=0$ 时，每个 LTL 公式均可以表达成这个形式。

我们的目标是将 $\eta(s)$ 精化为更简单（简短）的子公式 ν_i，直到所有的 ν_i 要么是命题变量要么是否定命题变量。

考虑一下，例如节点 s 包含一个子公式 $\nu_i = \mu \cup \psi$。我们将使用如下直到操作符“$\cup$”的性质：

$$\mu \mathbin{U} \psi = \psi \vee (\mu \wedge \bigcirc \mu \mathbin{U} \psi)$$

(见5.3节的公理A7)。表达式右边包含所定义公式自身，因此这可以被看做“U”操作符的不动点或是循环定义。因此，有两种满足 $\mu \mathbin{U} \psi$ 的方法。一种是满足直到操作符右边部分，即 ψ。另一种是延迟 ψ，意味着 μ 必须满足当前的后缀，同时 $\mu \mathbin{U} \psi$ 满足下一个状态。因为这两种不同的方法（或者等价地说，析取操作符“∨”的存在），我们将当前节点一分为二。在第一个备份中，我们添加一个 ν_i 子公式 ψ。在第二个备份中，我们添加一个 ν_i 子公式 μ 以及一个 κ_j 子公式 $\mu \mathbin{U} \psi$。其他公式类型也以类似方式处理，见后面表6.1。

当拆分一个节点时，我们保留 ν_i 和 κ_j 公式中的余项，以及前驱节点的列表。我们将已精化后的子公式 ν_i 和还未处理的部分分离。在完成精化过程或是将节点 s 一分为二后，生成它的后继 s'，其中 s' 的公式 ν_i 设置为 s 的 κ_j 公式，而 s' 的 κ_j 公式的集合为空。

在生成节点集合后，需要赋值接受条件。这要求必须满足等式右侧的每个直到子公式。也就是说，公式 $\mu \mathbin{U} \psi$ 的每个直到子公式必须被某个后缀 ξ^i 满足，一定存在一个后缀 $\xi^j \models \psi$，其中 $j \geqslant i$。（需要注意的是 $(\mu \mathbin{U} \psi) \rightarrow (\Diamond \psi)$。）重要的是，在先前介绍的节点拆分中，满足 $\mu \mathbin{U} \psi$ 的要么满足 ψ，要么延迟满足 $\mu \mathbin{U} \psi$ 到下一个状态（当 μ 满足当前后缀时）。我们需要保证并未永远延迟满足 ψ。为此，对每个该类型的子公式我们添加了一组广义 Büchi 条件。相应于形如 $\mu \mathbin{U} \psi$ 的子公式，每个条件是包含下列所有节点的集合：

- 包含 $\nu_i = \psi$
- 不包含 $\nu_i = \mu \mathbin{U} \psi$

现在，我们正式给出算法。为了应用如下的转换算法，我们首先将公式 φ 转换为*否定范式*(negation normal form)，其中否定只用于命题变量。我们首先使用布尔等价使得只保留布尔运算符 and（“∧”）、or（“∨”）和 not（“¬”）。在此基础上再加入否定操作，使其只出现在 AP 中的命题变量之前。可以依据 LTL 等价关系完成相应的转变：$\neg \bigcirc \mu = \bigcirc \neg \mu$，$\neg(\mu \vee \eta) = (\neg \mu) \wedge (\neg \eta)$，$\neg(\mu \wedge \eta) = (\neg \mu) \vee (\neg \eta)$，$\neg \neg \eta = \eta$，$\neg(\Box \psi) = \Diamond \neg \psi$，$\neg(\Diamond \psi) = \Box \neg \psi$，$\neg(\mu \mathbin{U} \eta) = (\neg \mu) \vee (\neg \eta)$ 以及 $\neg(\mu \vee \eta) = (\neg \mu) \mathbin{U} (\neg \eta)$。最后，我们使用等价关系 $\Diamond \eta = true \mathbin{U} \eta$ 和 $\Box \eta = false \vee \eta$，分别将操作符*终将*（“◇”）和*总是*（“□”）替代为*直到*（“U”）和*释放*（“∨”）。

此后，我们将假设 φ 已经属于范式。需要注意的是，转换 LTL 公式到范式并不会导致存储大小的显著（大于线性的）增长。

例如，考虑范式 $\neg \Box(\neg A \rightarrow (\Box B \wedge \Box C))$，我们用析取替换蕴涵，得到 $\neg \Box(A \vee (\Box B \wedge \Box C))$。加入否定操作，得到 $\Diamond(\neg A \wedge ((\Diamond \neg B) \vee (\Diamond \neg C)))$。然后，我们去除◇运算符，从而生成 $true \mathbin{U} (\neg A \wedge ((true \mathbin{U} \neg B) \vee (true \mathbin{U} \neg C)))$。

图节点是算法的基本数据结构（见图6.8）。一些节点被用做待构建自动机中的状态，其余的将在转换中删除。一个图节点包含下列的域：

Name：节点的唯一标识符。

Incoming：包含指向当前节点的入边节点的标识符列表。

New，*Old*，*Next*：每个域是一个 φ 的子公式集合。每个节点代表某个执行后缀的时序属性。在算法的上述非正式解释中，$New(s) \cup Old(s)$ 是 ν_i 公式。$New(s)$ 集合包含还未处理的子公式，而 $Old(s)$ 包含已处理的子公式，$Next(s)$ 中的子公式是上述解释中的 κ_j 公式。

我们保持构造完成的节点集合 *Nodes_Set*，这些节点组成了待构建自动机中的状态。初始情况下，*Nodes_Set* 集合为空。

将 LTL 公式转换为广义 Büchi 自动机的算法如图6.7所示。

```
    LTL translation algorithm

1   record graph_node = [Name:string, Incoming:set of string,
2         New:set of formula, Old:set of formula, Next:set of formula];
3   function expand (s, Nodes_Set)
4      if New(s)=∅ then
5         if exists node r in Nodes_Set with
6            Old(r)=Old(s) and Next(r) =Next(s)
7         then Incoming(r) = Incoming(r)∪Incoming(s);
8            return(Nodes_Set);
9         else return(expand( [Name⇐new_name(),
10           Incoming⇐ {Name(s)}, New⇐Next(s),
11           Old⇐ ∅ , Next⇐ ∅], Nodes_Set∪{s}))
12     else
13        let η ∈New(s);
14        New(s) := New(s)\{η}; Old(s) := Old(s)∪{η};
15        case η of
16        η = A, or ¬A, where A proposition, or η = true, or η =false=>
17           if η =false or ¬η ∈Old(s) then return(Nodes_Set)
18              else return(expand([Name⇐Name(s), Incoming⇐Incoming(s),
19                 New⇐New(s), Old⇐Old(s), Next⇐Next(s)], Node_Set));
20        η = μUψ, or μVψ, or μ ∨ ψ =>
21           s1:=[Name⇐ Name(s), Incoming⇐Incoming(s),
22              New⇐New(s)∪({New1(η)}\Old(s)),
23              Old⇐Old(s), Next=Next(s)∪{Next1(η)}];
24           s2:=new_node([Name⇐new_name(),
25              Incoming⇐Incoming(s),
26              New⇐New(s)∪({New2(η)}\Old(s)),
27              Old⇐Old(s), Next⇐Next(s)]);
28           return(expand(s2, expand(s1, Nodes_Set)));
29        η = μ ∧ ψ =>
30           return(expand([Name⇐Name(s), Incoming⇐Incoming(s),
31              New⇐New(s)∪({μ, ψ}\Old(s)),
32              Old⇐Old(s), Next=Next(s)], Nodes_Set))
33        η = ○μ =>
34           return(expand([Name⇐Name(s), Incoming⇐Incoming(s),
35              New⇐New(s), Old⇐Old(s),
36              Next⇐Next(s)∪{μ}], Nodes_Set))
37  end expand;
38  function create_graph (φ)
39     return(expand([Name⇐new_name(), Incoming⇐ {init},
40        New⇐ {φ} , Old⇐ ∅, Next⇐ ∅], ∅))
41  end create_graph;
```

图 6.7 LTL 转换算法

算法第 9、24 和 39 行中用到的函数 new_name() 为每个后继调用生成一个新名字。函数 New1(η)、New2(η)、Next1(η) 和 Next2(η) 的定义如表 6.1 所示。

为了转换公式 φ，算法从包含唯一入边的单个节点开始（第 39～40 行），这条入边始于一个特殊的伪节点 init。另外，有 $New = \{\varphi\}$，$Old = Next = \varnothing$。例如，图 6.8 是算法构建 $A \cup (B \cup C)$ 自动机的初始节点。

表 6.1 LTL 转换算法中的拆分表

η	New1(η)	Next1(η)	New2(η)	Next2(η)
$\mu \cup \psi$	$\{\mu\}$	$\{\mu \cup \psi\}$	$\{\psi\}$	$\varnothing$
$\mu \vee \psi$	$\{\psi\}$	$\{\mu \vee \psi\}$	$\{\mu, \psi\}$	$\varnothing$
$\mu \vee \psi$	$\{\mu\}$	$\varnothing$	$\{\psi\}$	$\varnothing$
$\mu \wedge \psi$	$\{\mu, \psi\}$	$\varnothing$	—	—
$\bigcirc\mu$	$\varnothing$	$\{\mu\}$	—	

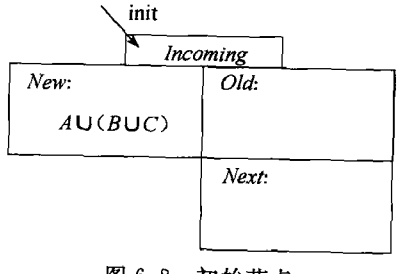


图 6.8 初始节点

对当前节点 s，算法检查 s 的域 *New* 中是否存在一个待处理的子公式（第 4 行）。如果不存在，则当前节点的处理完成。然后算法检查是否应该将当前节点加入 *Nodes_Set* 中；如果在 *Nodes_Set* 中存在一个节点 r，其中 r 的 *Old* 和 *Next* 域中的子公式和 s 的相同（第 5～6 行），那么不再需要节点 s（它的空间可以被再利用）。此时，将 s 的入边集合加入 r 的入边集合中（第 7 行）。否则，如果 *Nodes_Set* 中不存在一个这样的节点，将 s 加入 *Nodes_Set*，生成一个新的当前节点 s' 如下（第 9～11 行）：

- 存在从 s 到 s' 的边，即 s' 是 s 的后继。
- s' 的域 *New* 初始为 s 的 *Next*。
- s' 的域 *Next* 和 *Old* 初始设置为空。

图 6.11 是一个生成新的当前节点的示例。

每个全括号内的时序公式都有一个主模态（main modal）或是主布尔算子。这个模态或是算子出现在最外层的括号内（而公式在全括号内）。例如，公式 $(\Box(A \vee (B \wedge C)))$ 的主模态是“□”。公式 $(A \vee ((\Box B \wedge C)))$ 中的主运算符是“∨”。

如果 s 的 *New* 域非空，则 *New* 中的一个公式 η 将被选中（第 13 行），并从 *New* 中移除。根据 η 的主模态或是布尔算子，节点 s 被拆分为两个副本 s_1 和 s_2（第 20～28 行），或者演化（精炼）为一个新版本 s'（第 16～19 行以及第 29～36 行）。形成新节点首先要为节点选择新名字，并且复制 p 的域 *Incoming*、*Old*、*New* 和 *Next* 中的内容。然后将 η 加入 *Old* 的公式集合中。此外，根据下列不同情况，添加公式到 s_1 和 s_2，或 s' 的域 *New* 和 *Next* 中：

η 是一个命题，一个命题的否定，或一个布尔常量。如果 η 是 *false*，或 *Old* 中有 $\neg\eta$（我们认为 η 与 $\neg\neg\eta$ 相等），由于包含矛盾而无法被满足，当前节点将被舍弃（第 16～19 行）。否则，节点 s 如上所述演化为 s'。

$\eta = \mu \cup \psi$　节点 s 被拆分为两个副本 s_1 和 s_2（第 20～28 行）。对第一个副本 s_1，将 μ 加入 *New* 且 $\mu \cup \psi$ 加入 *Next*。在此拆分和其他任何拆分中，第一个副本可以重用旧节点 s 的空间。对第二个副本 s_2，将 ψ 加入 *New*。拆分是根据 $\mu \cup \psi$ 等价于 $\psi \vee (\mu \wedge \bigcirc(\mu \cup \psi))$ 进行的。例如，将图 6.8 中的节点拆分得到图 6.9 中的两个节点。其中 $\eta = A \cup (B \cup C)$，即 $\mu = A$ 和 $\psi = B \cup C$。进一步拆分图 6.9 中的右节点得到图 6.10，此时 $\eta = B \cup C$，因此 $\mu = B$ 和 $\psi = C$。

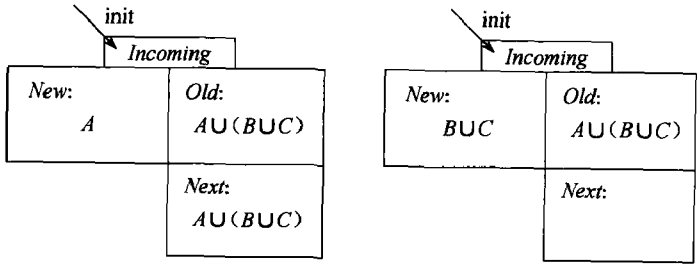


图 6.9　拆分图 6.8 中的节点

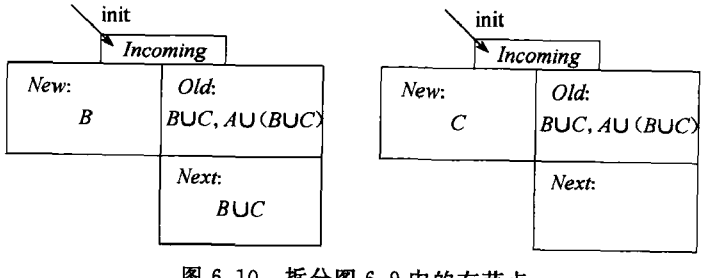


图 6.10　拆分图 6.9 中的右节点

$\eta=\mu \vee \psi$ 节点 p 被拆分为两个副本 s_1 和 s_2(第 20～28 行)。然后将 ψ 加入 s_1 和 s_2 的 New，μ 加入 s_1 的 New，$\mu \vee \psi$ 加入 s_2 的 $Next$。此处拆分是根据 $\mu \vee \psi$ 等价于 $\psi \wedge (\mu \vee \bigcirc(\mu \vee \psi))$ 进行的。

$\eta=\mu \vee \psi$ 然后，p 被拆分为两个副本 s_1 和 s_2(第 20～28 行)。将 μ 加入 s_1 的 New，ψ 加入 s_2 的 New。

$\eta=\mu \wedge \psi$ 然后，s 演化为 s'(第 29～32 行)。为了满足 η，需要同时满足两个子公式 μ 和 ψ，将 μ 和 ψ 都加入 s'的 New。

$\eta=\bigcirc\mu$ 然后，s 演化为 s'(第 33～36 行)，将 μ 加入 s'的 $Next$。

然后算法将递归地扩展新生成的副本。

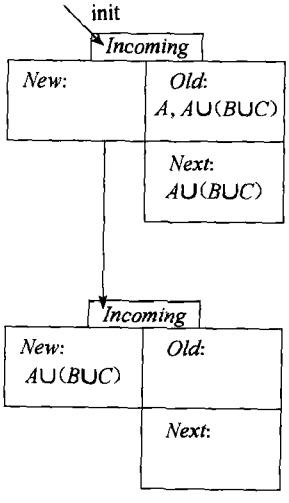


图 6.11 生成后继节点

由如上算法构建的 $Nodes\_Set$ 节点集合，现在可以转换为一个广义 Büchi 自动机 $\mathcal{B}=\langle \Sigma, S, \Delta, I, L, F\rangle$，其中：

- 字母表 Σ 包含被转换公式 φ 的命题 AP 集合中否定和非否定命题的合取命题公式。
- 状态集合 S 由 $Nodes\_Set$ 中的节点组成。
- $(s, s') \in \Delta$，其中 $s \in Incoming(s')$。
- 初始状态集合 $I \subset S$ 是包含特殊入边 init 的节点集合。
- $L(s)$ 是 $Old(s)$ 中否定和非否命题的合取命题。
- 广义 Büchi 自动机的接受条件中，$\mu \cup \psi$ 公式的每一个子公式有一个单独的状态集合 $f \in F$，使得 $\psi \in Old(s)$ 或 $\mu \cup \psi \notin Old(s)$ 的所有状态 s 都包含于 f。

图 6.12 是由 $A \cup (B \cup C)$ 公式所构建的节点组。图 6.13 是相应的广义 Büchi 自动机，包含两个接受集合，分别对应 $B \cup C$ 和 $A \cup (B \cup C)$。

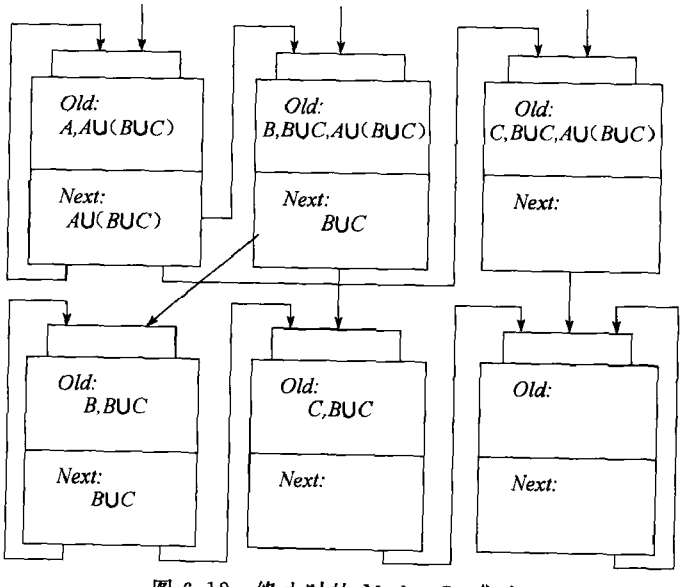


图 6.12 终止时的 $Nodes\_Set$ 集合

该算法构建的节点数和时间复杂度随公式大小的增长呈指数级增长。然而，经验显示生成的自动机通常较小。观察可知不必在 Old 中记录形如 $\mu \vee \nu$，$\mu \wedge \nu$ 和 $\bigcirc\mu$ 的子公式，除非这些公式出现在直到子公式的等式右侧，这样可以改进上述转换算法。上述限制是因为直到子公式的等式右侧用于定义接受条件。

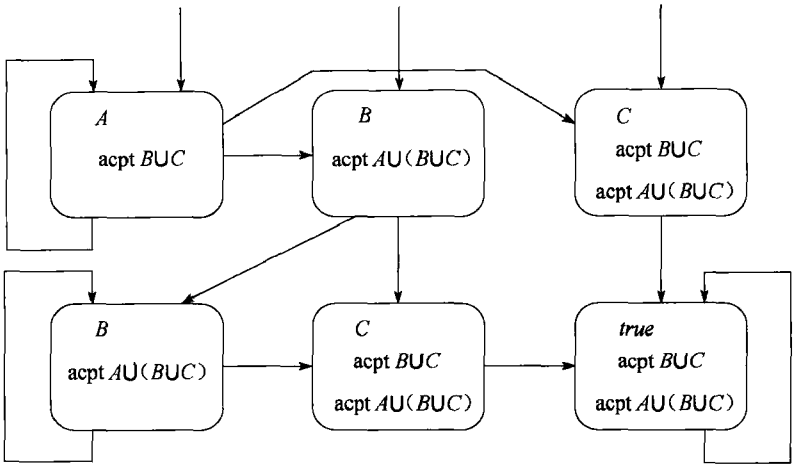


图 6.13 包含接受节点的自动机

基于状态间关系的另一个改进称为*互模拟*（bisimulation）。直觉上，将一对标签和接受条件一致的状态相关联，并且这对状态的后继也相互关联。再进一步，这些后继的后继也必须有一致的标签和接受条件并且相互关联，如此继续。互仿真的形式化定义将在 8.5.3 节给出，最简化的互仿真算法在 8.8 节给出。

回顾一下 6.3 节中的等式（6.2），在模型检验中我们用自动机表示坏行为，即不被规约 φ 允许的行为。先变换 φ 为一个自动机，再计算它的补集可能会导致双倍指数级大小增长。一个更好的解决方案是先通过在 φ 前添加否定符号计算 φ 的否定，然后将否定规约 $\neg\varphi$ 转换为自动机。此时最差可能引起单个指数级爆炸，而在实际情况中，如上转换产生的自动机结果通常很小。

6.9 模型检验的复杂度

为了判定模型检验的复杂度，我们首先需要决定如何衡量输入的大小。模型检验算法的两个输入是系统模型和属性规约。它们可以通过多种方法表示，而各个表示方法的规模差别很大。

例如，属性规约可以用 Büchi 自动机（接受不被允许的序列）、LTL，或是一元一阶或二阶逻辑表示（见 5.8 节）。模型检验的复杂度与规约大小存在着函数关系，因此它紧密依赖于选择何种规约：对于 Büchi 自动机是多项式级时间复杂度（事实上，可以在非确定对数级空间复杂度内完成），对于 LTL 表示法是 PSPACE，而对于一元一阶或二阶逻辑是 NONELEMENTARY。

这直接提出了如何选取规约机制以获得最小模型检验复杂度的问题。如 5.8 节所述，减小此复杂度却有可能伴随着增加规约大小的开销。例如，使用 Büchi 自动机的模型检验与使用同样大小的 LTL 规约的模型检验相比，前者的速度指数级快于后者。

另一方面，LTL 规约大小指数级小于相应的 Büchi 自动机表示。类似地，检验一元一阶逻辑规约 NONELEMENTARY 级慢于检验相同大小的 LTL 规约。另一方面，一元一阶逻辑规约的大小可能显著小于相应的 LTL 表示方法 [134]。

实际中，一个好的规约机制需要满足：许多常用的规约相对简短且易于检验。软件和硬件中一般常见属性的 LTL 规约通常都较为简短（尽管某些情况中，表示相同规约时一元一阶公式比 LTL 公式要简短得多），易于理解，并且使用广泛。相应地，我们选择 LTL 作为规约机制来进行模型检验的复杂度分析。

关于模型检验的复杂度我们还需要考虑的参数是系统模型的大小。假设一个并发系统中有 n 个独立的组件（进程）P_1，…，P_n，如 4.9 节中定义，其中 P_i 有 m_i 个本地状态，使用并发运算

符“‖”连接它们。因此，系统每个状态都是 n 个本地状态的组合。因此，全局状态的数目受限于乘积 $\prod_{i=1}^{n} m_i$。如果所有进程的大小都等于 m，那么状态空间中最多有 m^n 状态，也就是说，这里状态空间与进程数之间是指数级关系。在本章描述的模型检验算法中，我们在全局状态空间大小的多项式级时间内完成验证（或是非确定对数级空间）。

然而，使用全局状态空间衡量输入大小是非常粗略的估计方法：系统的自然描述是使用某种编程语言写成的代码，它与组件的大小成比例，即 $\sum_{i=1}^{n} m_i$，而不是与全局状态空间大小相关。因此，我们将被验证模型的大小估算为本地自动机大小的和，而不是乘积。简单假设所有进程都拥有相同的状态数 m，那么，一个时间复杂度为全局状态空间的多项式级（即上界 m^n）的算法实际上是其输入大小的指数级（即 $m \times n$）。

接下来，介绍 LTL 模型检验复杂度的一个相当理论化的分析，读者可以直接跳过本节。为了说明 LTL 模型检验复杂度的上界，我们先简单介绍另一个与 6.8 节中所述不同的将 LTL 转换为 Büchi 自动机的转换方法 [144]。在这种转换方法中，对于属性 φ 自动机中每个状态分别包含一个*一致性组件*和一个*活性组件*。

此处，我们不再使用 6.8 节变换中给出的否定范式，而是使用另一种范式。我们假设待转换公式 φ 中不包含*释放*运算符“V”。如果包含，则使用等价关系 $\psi_1 V \psi_2 = \neg((\neg\psi_1) U (\neg\psi_2))$ 移除它，公式大小只会增加一个常量因子。类似地，公式中也不包含*总是*模态算子“$\Box$”（其中 $\Box\psi = \neg\Diamond\neg\psi = \neg(true\ U \neg\psi)$）。

φ 的闭包，记为 $cl(\varphi)$，包含 φ 的子公式和它们的否定。对于形如 $\psi U \eta$ 的*直到*公式，闭包同样包括 $\bigcirc(\psi U \eta)$ 及其否定。我们将使用时序等价表示 $\neg\bigcirc\psi$ 为 $\bigcirc\neg\psi$，$\neg\neg\varphi$ 为 φ。对于 $cl(\varphi)$ 中的每个公式，每个一致性组件仅包含 ψ 和 $\neg\psi$ 中的一员。因此，对于 $\varphi = A U (B U C)$，$cl(\varphi)$ 包含 A，$\neg A$，B，$\neg B$，C，$\neg C$，$B U C$，$\neg B U C$，$\bigcirc(B U C)$，$\bigcirc\neg(B U C)$，$A U (B U C)$，$\neg(A U (B U C))$，$\bigcirc(A U (B U C))$，$\bigcirc\neg(A U (B U C))$。

此外，一致性组件必须满足下列限制：

- 如果包含*直到*公式 $\mu U \psi$，那么它必须包含 ψ，或是同时包含 $\bigcirc(\mu U \psi)$ 和 μ。
- 如果包含形如 $\mu V \psi$ 的公式，那么它必须包含 μ 或 ψ。
- 如果包含形如 $\mu \wedge \psi$ 的公式，那么它必须同时包含 μ 和 ψ。

当一个状态 r 是状态 s 的后继时，需满足：如果 s 的一致性组件包含形如 $\bigcirc\psi$ 的子公式，那么 r 的一致性组件必须包含 ψ。因此，一致性组件保证了相邻状态间的一致性。

每个状态的活性组件包含形如 $\mu U \psi$ 的 ψ 中*直到*子公式的一个子集。考虑状态 s 和后继 r，如果 s 的活跃性组件为空，那么 r 的活性组件将包含出现在 r 的一致性组件中的所有*直到*子公式。否则（当 s 的活性组件非空时），对每一个在 s 的活性组件中出现的 $\mu U \psi$ 公式，仅当 s 的一致性组件中没有出现 ψ 时，$\mu U \psi$ 必须出现在 r 的活性组件中。直观而言，活性维护了仍需被满足的*直到*子公式的集合。只要从某状态开始 $\mu U \psi$ 子公式被满足，由于相关状态满足了 ψ，那么它将从活性组件中被移除。当所有*直到*子公式都被满足，即活性组件为空，将从一致性组件复制需要被满足的*直到*子公式集合。如果 $cl(\varphi)$ 中不存在任何*直到*公式，那么所有状态都是接受状态。

初始状态中的一致性组件包含待转换公式 φ 自身，其活性组件为空。

6.8 节介绍的转换产生的自动机 [49] 比这里描述的要小很多。然而，使用本节中为了复杂度分析而给出的替代算法的优点是：可以在多项式空间（和时间）内检验（1）给定的状态是否属于转换；（2）根据上述约束给定状态是否是另一状态的后继。这个检验过程不需要先完成所有

节点和边的构建。需要注意的是转换中每个状态的大小是待转换 LTL 公式大小的多项式级，尽管状态数量是指数级。

现在，我们可以在未完成相关自动机构建的情况下对属性自动机与系统自动机的笛卡儿乘积状态空间进行搜索。这个搜索是为了寻找从初始状态可达的接受状态 s，并且从 s 可以抵达自身。如果存在一个这样的状态，那么系统不满足它的规约。为了空间有效性，我们使用*二分搜索法*。递归地描述如下：为了寻找从状态 s 到状态 r 是否存在一条长度不大于 l 的路径（简单假设 l 为 2 的幂次），我们需要寻找状态 q 使得存在（1）一条从 s 到 q 长度不大于 $l/2$ 的路径；（2）一条从 q 到 r 长度不大于 $l/2$ 的路径。当长度为 1 时，检验是否存在从 s 到 r 的路径意味着检验 r 是否是 s 的后继，而这个已知可在多项式时间内完成。

在二分搜索中需要的最大状态存储数目是 $\log l$，其中每个状态的大小是 φ 和进程数 n 的多项式级。路径的最大长度是自动机状态空间大小（m^n）与 φ 的 Büchi 自动机大小（$4^{|\varphi|}$，其中 $|\varphi|$ 是公式 φ 的长度）的乘积。每个状态的大小是 $n\times\log m$ 和 $|\varphi|$ 的多项式级。简单计算可知二分搜索的总体空间复杂度是 $|\varphi|$ 和 n 的多项式级。由此可知，模型检验的复杂度上界为 PSPACE。

对于被建模系统的大小，由 Kozen [77] 可知，复杂度下界是 PSPACE。这样，即使是检验自动机集合的交集中某个状态的可达性检验之类的简单属性，也是 PSPACE 完全的。Clarke 和 Sistla [132] 指出，相对于待验证的 LTL 属性的大小，模型检验也是 PSPACE 完全的。因此，LTL 属性的模型检验对于系统大小和待验证属性大小都是 PSPACE 完全的。

6.10 表示公平性

如果希望在公平性假设下完成模型检验，就需要对模型检验算法进行相应的调整。

在用 LTL 描述的公平性假设 ψ 下，检验系统 S 是否满足性质 φ 的一种方法，是证明 S 满足 $\psi\rightarrow\varphi$。这就是说，S 的每一个执行在 ψ 下是公平的并且满足 φ。这并不是检验公平性的一种有效方法，因为公平性公式 ψ 可能很大，将其转换后可能产生一个相当大的 Büchi 自动机。

在公平性下检验属性的另一种方法是修改验证算法 [88]。例如，考虑弱转换公平性的情况。一个公平的执行是其中每一个转换都无限多次不可执行或是被执行无限多次（或是两者都有）。为了检验这个，假设不同的转换都有各自唯一的名字。我们在每条边上标记相应的转换名。现在可以检验由 Tarjan 的 DFS 算法获得的每个可达的强连通组件。我们称这样的一个组件包含一个反例，仅当针对其中每个转换 t 它包含下列两者之一：

- *存在一条标记为 t 的边。*
- *存在一个 t 不可执行的状态。*

我们可以证明：如果存在一个这样的强连通组件，就存在一个公平性反例。这个反例将循环地（允许重复）通过这个组件中所有的状态和边。事实上这给出了一个无限重复的反例。这样，对每一个转换 t 将有无限多个状态，其中要么 t 不可执行，要么 t 出现无限多次。反之，如果存在一个公平性反例，那么它必须是满足上述条件的强连通组件中的某一部分。

类似的算法也出现在其他的公平性假设中，包括在 4.12 节中提到的算法。我们可以定义比简单 Büchi 接受条件更为复杂的接受条件（但可以通过相同的语言集合来描述）[141]。这样的接受条件可以用于表示系统自动机的多种公平性条件 [3]。

6.11 检验 LTL 规约

LTL 转换算法和 Büchi 自动机的空集检验算法可以用于检验 LTL 公式的有效性。回顾一下，一个 LTL 命题公式是有效的，当且仅当在公式命题集合上它对每个可能的序列都满足。这对检验是否一个规约 φ 比另一个 ψ 更强（$\varphi\rightarrow\psi$ 是否有效）很有帮助。此时，如果我们验证一个系统

满足 φ，那么它也满足 ψ。

我们同样关注于检验某个规约 φ 是否被至少一条序列满足，即它不包含内部矛盾。如果没有任何执行序列满足这个属性，那么相应的规约显然是无用的。

有效性和可满足性之间的关系显而易见：一个公式 φ 是有效的，仅当 $\neg\varphi$ 不被满足。因此，检验一个公式 φ 是否有效可以通过添加否定符号的前缀并检验 $\neg\varphi$ 是否不被满足来完成。

这样，我们约简（转换）公式 φ 的有效性判定问题为一个我们已知如何解决的问题：

1. 将 $\neg\varphi$ 转换为自动机 $\mathcal{A}$，如 6.8 节所述。

2. 检验 $\mathcal{A}$ 的语言 $\mathcal{L}(\mathcal{A})$ 是否为空，如 6.6 节所述。

$\mathcal{A}$ 的语言恰是满足公式 $\neg\varphi$ 的执行集合。如果为空，那么 $\neg\varphi$ 不被满足，即 φ 是有效的。

有时我们会观察到转换算法可能生成比 $\neg\varphi$ 指数级大的自动机。而空集检验的复杂度是自动机大小的线性级别。因此上述算法的复杂度是公式大小的指数级。事实上，我们利用 6.9 节所描述的算法，使用二分搜索技术能够更高效地（从空间复杂度层面上）检验（不）可满足性。其中，对 $\neg\varphi$ 的自动机状态空间的二分搜索并不需要预先完整构建整个自动机 [144]。检验 LTL 规约的（不）可满足性的复杂度也同样是 PSPACE 完全的 [132]。

6.12 安全属性

时序属性的一个重要类别称为*安全属性*（safety property）。Lamport [83] 非正式地描述安全属性为“永远没有坏的事情发生”。试图形式化地描述这一重要属性的尝试很多，如使用拓扑逻辑或线性时序逻辑的过去模态。这里，我们将采用 Alpern 和 Schneider [5] 的基于自动机的定义。描述安全性的简单结构自动机可以立即被转换为模型检验算法 [131]。

为了直观地说明安全性质，首先考虑一个不变式的简单案例。一个不变式必须被整个执行序列满足。相应地，可能发生的“坏”事情就是我们到达了某个不满足不变式的状态。例如，可能有一个状态，它的两个进程都同时在临界区内。因此，一个不变式是一个安全属性。然而，安全性可能比这个例子更为普遍。在非正式定义的安全性中，坏事情并不必是程序执行序列中的一个坏状态的结果。它可能对应于某个顺序的有限状态序列，例如，一个消息在其他消息前到达消息队列，却在之后被读取。

我们可以建立一类自动机，用于识别安全属性的否定。对于一个给定的安全属性，该自动机将发现执行中违反安全性的前缀。这样的前缀将包含坏事情已经发生的证据，从而，整个执行序列可以被认定为坏的而被丢弃。这个前缀的运行中最后一个状态将会被标记为坏状态从而与其他状态相区别。需要注意的是，属性自动机的状态对应于自动机在输入字有限前缀上的运行中收集的信息。在这个自动机的运行中，当进入一个坏状态时，整个执行被看做违反安全条件。因此，沿着到达坏状态的执行上的有限前缀的运行，足以识别待验证模型的执行序列违反了安全性。无论在这个（无限）执行上的（无限）运行如何继续，都已违反了安全性。

因此，将一个*安全自动机*定义为六元组 $\langle\Sigma, S, \Delta, I, L, B\rangle$。其中 Σ，S，Δ，I 和 L 的含义与 Büchi 自动机中相同。集合 $B\subseteq S$ 是坏状态的集合。我们称 $S\setminus B$ 中的状态是*好状态*。相关限制包括：

1. 一个坏状态没有任何后继。这是因为，当到达坏状态时，已经违反了安全性。对于安全属性，发现违反该属性的执行前缀已经足够。

2. 令 q 是一个好状态，q_1，…，q_n 为其直接后继，且 $L(q_1)\vee\cdots\vee L(q_n)=true$。因而，如果没有违反安全性，就不能对这个执行作出判断，这样也就总是存在一个后继自动机状态。

安全自动机可以是非确定的，即使对每个安全属性都存在对应的确定性自动机（可能大于指数级）。自动机运行的定义类似于 Büchi 自动机。唯一的不同是一个执行违反安全属性，仅当

存在一个运行在有限前缀后到达 B 中的某个坏状态。需要注意在这种情况中，由于 B 中状态没有后继，因此运行是有限的。一个安全自动机$\overline{\mathcal{B}}$可以识别违反安全属性的执行前缀（因此，用$\overline{\mathcal{B}}$表示自动机，而不是$\mathcal{B}$）。在非确定性安全自动机中，即使存在其他从不到达坏状态的运行，一个到达坏状态的运行已足够判定执行违反安全属性。

我们可以根据以下步骤转换安全自动机到 Büchi 自动机：添加一个状态 l 到安全自动机，并标记为 *true*。这个状态包含一个自循环。对每个坏状态添加一条到 l 的边。状态 l 是这个自动机的唯一接受状态，同时坏状态可以被忽略。

这样构造出的安全自动机允许我们简化模型检验算法。我们希望找到违反安全属性的执行的有限前缀，其中存在同时属于系统自动机$\mathcal{A}$和安全自动机$\overline{\mathcal{B}}$的有限序列。$\mathcal{A}$和$\overline{\mathcal{B}}$的交集定义如下：交集的每个状态是包含$\mathcal{A}$中状态和$\overline{\mathcal{B}}$中状态的一组状态。交集的初始状态是$\mathcal{A}$和$\overline{\mathcal{B}}$的初始状态组。转换关系将一组状态与另一组相关联，使得组件$\mathcal{A}$执行一个$\mathcal{A}$中的转换，组件$\overline{\mathcal{B}}$执行一个$\overline{\mathcal{B}}$中的转换。交集中的状态是坏状态，仅当其$\overline{\mathcal{B}}$组件是坏状态。

用一组坏状态替换 Büchi 条件给出了更简单的模型检验算法。通常，我们在可达强连通组件中搜索一个可达的 Büchi 接受状态，然后构建一个反例，它包含一个经过接受状态的循环的有限前缀。这里，我们寻找从交集中初始状态可达的坏状态。作为反例，由于错误已经出现在前缀中，因此足够给出从初始状态到坏状态的前缀。因此，一个简单的“随时”DFS 足以搜索出这样的状态。值得注意的是，此方法不需要计算交集图的强连通分量。

6.13 状态空间爆炸问题

模型检验的主要挑战是缓解状态空间爆炸问题。这个问题反映出，由一组并发组件构成的系统的状态数，是各个并发组件的状态数的乘积。如果一个系统包含 n 个相同的组件，每个组件有 m 个状态，且进程间没有通信或不使用共享变量交互，那么有 m^n 个状态（m 种可能性的 n 个独立选择）。当然，在更真实的情况中，进程间的交互可能显著限制可能的状态数，尽管如此，状态数目可能仍然居高不下。

有多种针对状态空间爆炸问题的策略，但没有一种能保证这个问题得到有效解决。经验和实验可以用来帮助寻找究竟应该使用哪种策略或是策略间的组合。许多模型检验工具最多包含其中一条策略。

我们将列出其中一些策略。感兴趣的读者可以在 [30] 中找到更多的细节信息。

二叉判定图（Binary Decision Diagram）[24]。应对状态空间爆炸的一种重要技术，避免分开计算和存储待验证系统中的每个状态。不再将每个状态作为一个单独的实体存储在状态空间中，而是在无环图（cycle free diagram，DAG）中保存状态的集合。合并同构子树（可以相互映射并保存后继关系和标签的子树）以节约空间。这种数据结构除了应用于深度优先搜索，还常用于其他搜索策略中。搜索过程回溯而不是前进。每一步计算时，一次性计算所有用当前 DAG 表示的状态前驱集合，而不是一个状态接着一个状态地计算。

偏序约简（Partial Order Reduction）[55，113，142]。这种方法允许只检验执行的一个子集，从而使用较小状态空间。这个方法利用了并发转换间的交换性。基于观察可知，如果将两次执行中相同的并发事件进行重排序后相互一致（即相同偏序执行所产生的交错序列，见 4.13 节），那么一般规约无法区分这两次不同的执行。

对称性（Symmetry）[43]。这个方法利用状态组件间的排列顺序。观察可知，在某些规约下（特别是硬件），不能区分通过排列不同进程的各种组件而获得的执行。因此，这种情况下，不需要遍历全部可能的状态，而是对在排列组合下等价的状态集合进行统一处理即可。

6.14 模型检验的优点

模型检验大部分是自动化的。为了将待验证系统表示为工具可以处理的形式，需要进行建模（尽管有些时候也可以自动完成）。可能需要抽象处理来约简验证问题的复杂度。此外，模型检验工具的使用者通常需要设定一些验证参数（例如，估算的状态数）。然而，相比于其他软件可靠性方法，验证过程中用户需要处理的部分相当小。

模型检验是基于相对容易实现的技术发展而来。现在已有大量的模型检验工具（部分列于本章末尾），这些工具使得我们可以检验很多种属性。此外，当模型检验工具发现系统不满足相关属性时，工具会提供相关重要信息，给出包含违反相应属性的执行作为反例，从而可以用于对系统进行调试。

6.15 模型检验的缺点

模型检验是主要由验证程序自动执行的一组算法集合。实际中，因为模型检验中产生的组合复杂度，用户必须常常提供各种参数以调整验证过程，如搜索栈大小、允许的存储器大小和模型变量间的顺序。这意味着，在很多情况中，模型检验工具的使用者是使用这种工具的专家，而在某些情况中就是工具的开发者本人。

模型检验，类似于演绎验证，通常要求首先建模待验证系统。模型检验工具通常包含一个内在的建模语言。某些时候，原始代码和验证工具使用的语法间的自动转换是可行的。然而，验证工具经常包含对内在语法的特定优化和启发式技术，在机器转换中这些信息都将丢失。而且，需验证的程序最初通常不是一个有限状态系统，因此进行模型检验前需要进行抽象。这个抽象过程通常是手动完成。结果，如果将模型检验用于程序的抽象模型，并发现了错误，我们通常需要手工确认这个错误是真实的，而不是原始程序和相应模型间不一致而产生的误报。

软件的自动验证是一个计算难题。它的复杂性，如6.9节所分析的，使它在不久之前被归类为无法解决的难题。硬件的迅速发展，以及针对模型检验的很多特殊的启发式技术，使得自动软件验证可以付诸实践，至少用于某些软件产品的抽象版本。然而，我们必须记住已知的复杂度问题依然存在，自动软件验证仍是个艰巨的任务，由于存储限制或时间限制而非常容易失败。

现有的自动验证工具经常无法完成验证任务，这在很大程度上依赖于特殊的启发式技术。甚至预测给定的工具何时成功或是失败也很难。原因之一是即使验证某个特定长度的程序，它们所需的状态空间大小之间也区别显著。

6.16 选择自动验证工具

根据经验法则，基于特定的观察和实验，在给定的技术集中可以指出一些执行性能更优的工具。例如，许多研究人员相信基于BDDs [24] 的工具更适于硬件电路验证，而基于“即时”技术 [34]、自动机理论 [144] 和偏序约简 [113] 的工具适于交错软件模型优化。然而，这些初步结论并不是一成不变的。

由于验证工具的“专门知识”通常由它们的特殊启发式技术来证明，因此专门针对某一类软件进行优化的验证工具并不罕见。例如，可以对验证工具进行专门优化来处理包含异步消息传递的程序，而共享变量的存在则可能抵消大部分优化效果。因此，不同工具使用的形式化间的自动转换通常比直接使用给定工具的专用语法要低效得多。因为转换过程通常会失掉特定目标工具优化后的特殊结构。此外，中立并全面的“基准”可以帮助我们在不同工具间作出选择。

6.17 模型检验项目

习题 6.17.1 再一次考虑4.6节中Dekker的互斥算法。

任务 1 将此算法转换为一种模型检验工具的建模语言（例如，使用下一节中列出的某个工具）。

任务 2 将下列属性形式化为相应工具的规约语言，并完成验证：

排他性：两个进程不能同时进入临界区。

活性：只要有进程试图进入其临界区（通过设置自己的变量 $c1$ 或 $c2$ 为 0），那么它最终总可以进入。

任务 3（进阶题） 注意以下的模型变体：

允许每个进程在非临界区有自循环。禁止这个自循环时，再一次检验。

对 *wait until turn*＝1，允许有以下两种不同的实现：

1. 忙等待：每个进程检验 *turn* 的值，直到它被另一个进程设置为允许继续前进的值。
2. 非忙等：每个进程自我冻结，直到其继续前进需要的值被其他进程设置。

验证上述属性的所有组合情况。检验算法是否满足活性属性。现在，通过附加弱公平性条件，强制它满足活性。并不是每一个工具都提供公平性，但某些工具允许规约公式 *fair*。这种情况中，不再检验属性 φ，而是等价地检验 $fair \rightarrow \varphi$。（将验证结果和 8.4 节中讨论的有趣现象进行对比。）

习题 6.17.2 考虑 4.15 节中的通信协议。使用一种可用模型检验工具中的形式化机制对其建模。检验习题 5.10.3 中的形式化属性。

6.18 模型检验工具

COSPAN 和 FORMALCHECK 系统可以从 Bell 实验室的 R. P. Kurshan 处获得，他的网站是：

http://cm. bell-labs. com/who/k/index. html

MURPHY 模型检验工具可以从 Stanford 大学获得，URL 如下：

http://sprout. stanford. edu/dill/murphi. html

SPIN 模型检验工具可以从 Bell 实验室获得，URL 如下：

http://netlib. bell-labs. com/netlib/spin/whatispin. html

SMV 模型检验工具可以从 CMU 获得，URL 如下：

http://www. cs. cmu. edu/~modelcheck/code. html

VIS 模型检验工具可以从 Berkeley 获得，URL 如下：

http://www-cad. EECS. Berkeley. EDU/Respep/Research/vis

需要注意的是，使用这些形式化方法需要填写并寄出相应的授权表格，并遵守相关规章制度。

6.19 扩展阅读

关于模型检验技术的最新综合性书籍如下：

E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, MIT Press 1999.

它提供了多种技术和启发式方法，并介绍了一些真实案例。

如下 Kurshan 的书中描述了基于自动机的模型检验方法：

R. P. Kurshan, *Computer Aided Verification of Coordinating Processes: the Automata-Theoretic Approach*, Princeton University Press, 1995.

还有一些书中介绍了基于 CTL 分支时序逻辑、使用 BDD 方法的模型检验技术。这个方法常被用于硬件验证，也可以在软件中应用。

K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Press, 1993.

Ch. Meinel, Th. Theobald, *Algorithms and Data Structures in VLSI Design*, Springer-Verlag, 1998.

演绎式软件验证

"禀奏陛下，"杰克说，"这不是我写的，它们不能证明是我写的：末尾没有签名。"

"要是你没有签名，"国王说，"只能说明情节更加恶劣。你一定是想干坏事，否则你就会像一个诚实的人那样签上你的名字。"

"当然，这就证明他有罪，"王后说。

刘易斯·卡洛尔《爱丽丝漫游奇境记》

20世纪60年代末，一些研究人员，特别是Floyd [45] 和Hoare [63]，积极倡导用形式化方法来验证算法和计算机程序的理念。由此产生了结合程序文本和逻辑的形式化机制，它们用于专门开发的证明系统中。之后的工作考虑了加入多种编程结构，如数组变量、过程调用 [85] 以及并发 [86, 108]。这些证明系统也可以被反过来用：根据一个给定的规约，使用与演绎证明相关的方法和工具来开发正确性可证的程序。最近，一系列强大的自动化的定理证明器被开发了出来。这些证明器有助于获得正确性证明，以及确保演绎证明方法被正确使用。

除了能增强程序的正确性外，演绎式程序验证还衍生出其他几种有益的方法。其中之一是使用不变式，就是使用需在代码执行中始终保持正确的断言。不变式能用来检查代码在不同的开发阶段的一致性。它们作为额外的运行时检查机制被插入到代码中，当有不变式不满足时，程序就会中断并给出警告信息。

软件可靠性证明系统的另一个用处是为不同的编程语言结构定义语义，人们能通过证明规则来理解相对应的结构，例如一个 while 循环。

因为演绎验证通常颇为冗长，因此在实际代码上并不经常使用。但它能用在基础的算法或是简化了的代码抽象上。有时简化、抽象一个程序的过程也能被形式化地证明。

7.1 流程图程序的验证

第一个用于程序验证的证明系统是由 Floyd [45] 开发的。该系统处理一类简单的由流程图表示的程序。这些程序包括简单的赋值语句和判定谓词，其中赋值语句形如 $v := e$，v 为某程序变量，e 为某表达式。表达式 e 是在给定的结构和签名上的一个一阶项。判定谓词是在该结构上的一个非限定的一阶公式。一个流程图有四类节点（参见图 7.1）：

- 有一条出边但无入边的椭圆代表开始（begin）语句。
- 有一条入边但无出边的椭圆代表终止（end）语句。
- 有一条或多条入边和一条出边的平行四边形代表赋值（assignment）语句。
- 有一条或多条入边和两条分别标记着真（true）和假（false）的出边的菱形代表判定（decision）语句。

图 7.1 是一个流程图程序的例子。

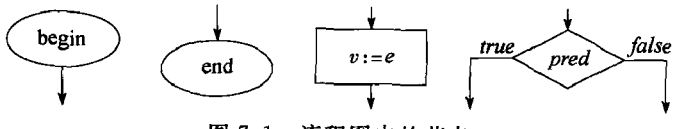


图 7.1 流程图中的节点

在计算科学的早期，程序员习惯于在编写实际代码前先画出算法的流程图。现在尽管流程图仍在使用，但多少已不如过去那么普遍。流程图也能通过编译的方式从编程语言中自动获取。为了能用合理的可视化形式来表示流程图，同时又尽量减少边的交叉，可以使用特殊的图布局算法 [50]。

为了简化标识，假设程序的初始值储存在一组输入变量 x_0，x_1，…中，且这些变量的值在程序的执行过程中不发生改变。因此，输入变量不会出现在赋值语句的左侧。

我们对每一个程序关联一个称为*初始条件*（initial condition）的一阶断言，该断言的自由变量皆为输入变量。初始条件规定了程序在每次执行开始时赋给这些变量的值。

程序的一个*状态*简单地说就是程序变量的一组赋值。我们说过：赋值 a（一个状态）满足公式 φ 可记为 $M_a(\varphi)=\mathrm{TRUE}$（这里未提及结构 $\mathcal{S}$，假设 $\mathcal{S}$ 可由上下文获得），或是记为 $a\models^{\mathcal{S}}\varphi$。程序的执行是一个有限或无限的状态序列，其中第一个状态满足初始条件，其余每个后继状态 b 都由其前驱状态 a 按如下方式获得：

- 若 a 是谓词为 p 的判定节点入边上的状态，b 是该节点标记为 *true* 的出边上的状态，且 $a\models^{\mathcal{S}}p$ 成立，则 b 与 a 相同。
- 若 a 是谓词为 p 的判定节点入边上的状态，b 是该节点标记为 *false* 的出边上的状态，且 $a\models^{\mathcal{S}}\neg p$ 成立，则 b 与 a 相同。
- 若 a 是赋值 $v:=e$ 前的状态，b 是紧跟该赋值后的状态，则 b 为 $a[T_a[e]/v]$。回顾 3.3 节，$T_a[e]$ 是表达式 e 根据状态 a 计算而得的值，$a[d/v]$ 是除了 v 值被设为 d 外其余都与 a 相同的一个赋值。因此，赋值之后的状态 b 除了 v 的值以外其余都与 a 相同，而 v 的值是表达式 e 根据之前的状态 a 计算而得的值。

极化节点（polarized node）是由一个流程图节点和其上的一条特别选定的出边组合而成。因此，任意的赋值语句构成一个极化节点，因为它仅有一条出边。每一个判定节点都构成两个极化节点：一个出边标记为 *true* 的*肯定*（positive）判定和一个出边标记为 *false* 的*否定*（negative）判定。

程序的*最终断言*（final assertion）一个一阶公式。若程序在用初始条件初始化后被执行，则每次执行后的最终状态都应满足最终断言。流程图程序的一个*位置*（location）是连接两个流程图节点的一条边。初始条件关联在开始节点的出口位置上，而最终断言关联在终止节点的入口位置上。初始条件和最终断言不过是众多关联在流程图程序位置上的断言中的两个。关联在流程图位置上的断言也称为*不变式*，因为当程序的控制点运行到不变式关联的地方时，这些断言要在该状态下始终成立。通常意义下不变式应在程序的执行过程中始终成立。关联在流程图位置 X 上的断言 φ 在以下情况下成为一个不变式：如果我们添加一个仅当程序运行到位置 X 时成立的命题 $at(X)$，则 $at(X)\rightarrow\varphi$ 在程序运行阶段始终成立。该式在 X 以外的程序控制点依然成立，因为 $at(X)$ 为 FALSE，因此整个蕴含式为 TRUE。

在节点入边上的不变式称为该节点的*前置条件*（precondition），而在节点出边上的不变式称为该节点的*后置条件*（postcondition）。注意每个判定节点有两个后置条件，而每个极化节点仅有一个。同一个不变式既可是一个节点的前置条件，又可是另一个节点的后置条件。在如图 7.2 所示的流程图程序中，关联在位置 A 上的初始条件是 $x1\geqslant 0\wedge x2>0$。关联在位置 F 上的最终断言是 $(x1\equiv y1\times x2+y2)\wedge y2\geqslant 0\wedge y2<x2$。我们能在流程图的位置（即流程图的边）关联额外的断言，例如在位置 C，我们能关联断言 $(x1\equiv y1\times x2+y2)\wedge y2\geqslant 0$。事实上为了获取正确性的证明，我们会在每一个流程图位置关联这样一个断言。当然，在一个位置上关联的断言并不立即就成为不变式。我们仍然需要证明关联的断言是不变式，证明的方法我们很快就会介绍。

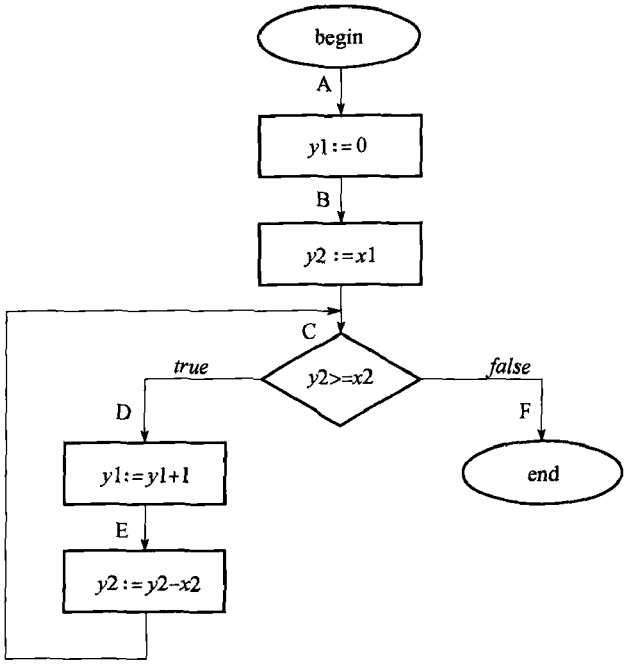


图 7.2　整数除法的流程图程序

我们可以把一个顺序程序 P 看做是输入状态和输出状态间的关系。因此，$P(a, b)$ 表示 P 从满足 a 的状态开始执行，在终止节点上终结时，停留在满足 b 的状态上。为简化叙述，我们假设 P 是确定的。因此，给定一个初始状态 a，P 要么不终结，要么终结时停留在唯一个最终状态 b 上。

程序的部分正确性（partial correctness）是一个三元组的断言：初始条件 φ、程序 P 和最终断言 ψ，通常记为 $\{\varphi\}P\{\psi\}$。该断言的含义是若程序从满足 φ 的状态运行，当其终结于终止节点时，最终状态满足 ψ。形式化地，对每一对程序状态 a，b，

若 $P(a, b)$ 且 $a \models_S \varphi$，那么 $b \models_S \psi$。

该形式化定义十分微妙。透过仔细的观察可以发现，即便程序从满足初始条件的状态运行，该定义也不保证程序的终止。这并不是巧合。事实证明，单独证明部分正确性比同时证明程序终止更为简单。事实上，“部分正确性”这个名字就代表了它并不能给出程序正确性的完整依据。*完全正确性*（total correctness）结合了部分正确性和终止性，我们将在稍后对其做介绍。部分正确性的形式化定义允许从满足初始条件 φ 的状态执行的程序不终止。

为证明部分正确性，我们需要表明若程序从满足初始条件的状态开始执行，当它执行到对应流程图某边的特定位置时，它满足那个位置的不变式。为证明此命题我们需证明注解流程图的不变式之间的一致性。令 c 为一极化节点，其前置条件为 $pre(c)$，后置条件为 $post(c)$。我们需证明如下命题：

若程序的控制点恰在 c 之前，其状态满足 $pre(c)$，当 c 执行后控制点移到标注有 $post(c)$ 的边上，那么此时的状态满足 $post(c)$。

这可以用之前的标识 $\{pre(c)\}c\{post(c)\}$ 来表示。对极化的流程图节点而言有三类*一致性条件*：

1. c 是肯定判定，换言之，$post(c)$ 是在取谓词 p 为 *true* 的出边后的后置条件，那么此判定前后的状态保持相同。所以，我们需要证明若 $pre(c)$ 在该状态下成立，且判定谓词的解释为

TRUE，那么 $post(c)$也成立。因此，我们必须证明蕴含式

$$(pre(c) \wedge p) \rightarrow post(c)$$

2. c 是否定判定，换言之，$post(c)$是在取谓词 p 为 $false$ 的出边后的后置条件。此时$\neg p$ 成立，其余与前述情况相同，我们必须证明蕴含式

$$(pre(c) \wedge \neg p) \rightarrow post(c)$$

3. c 是赋值语句 $v := e$。此情况较之前二者更难，因为赋值前后的状态是不同的。特别地，断言 $pre(c)$和 $post(c)$不是就相同的状态进行推论。因此首先应使这两个公式推断同一状态。我们可以相对化(relativizing) $post(c)$得到断言 $relpost(c)$，使之推断赋值前的状态。(我们也可以相对化 $pre(c)$来推断赋值后的状态，但这会带来更复杂的一阶公式，其过程涉及添加量化，参见[11]。)

当 $c=$ '$v := e$'，我们定义 $relpost(c) = post(c)[e/v]$。该构造一开始可能显得有点儿奇怪。它规定了用表达式 e 代替后置条件中的变量 v 的每个自由出现。若说赋值以正向的方式运作，那么该替换就以逆向的方式运作：从后置条件到前置条件！

欲了解为何该方法可行，考虑两组变量：一组代表了在赋值前状态下变量的值，另一组是这组变量的上撇号版本，代表了恰在执行赋值后的状态。两组变量的关系是，对每个不同于 v 的变量 u，$u' \equiv u$，其余 $v' \equiv e$。注意 e 仅用非加撇号的变量来表达，因为赋值表达式使用的是基于赋值前状态的值。

因此 $relpost(c)$能按如下方式得到：首先，用加撇号的变量重写 $post(c)$，用程序变量的加撇号副本替代其自由出现。我们现在需要写代表赋值前状态的公式。这可以根据上文中加撇号与非加撇号变量的关系，通过消除加撇号变量来达到。就是说，我们用非加撇号的版本来替代加撇号的变量，除了用 e 替代 v'。这给出了 $post(c)[e/v]$。注意在实际的替换中，首先将变量转换成它们的加撇号版本不是必需的，仅需用 e 替代 v，因为加撇号变量仅是为了说明逆向替换而引入的。

在部分正确性证明中，我们需要证明每个满足 $pre(c)$的状态也满足 $relpost(c)$。如果它满足了 $relpost(c)$，就保证了执行赋值后的结果状态满足 $post(c)$。该蕴含 $pre(c) \rightarrow relpost(c)$，可被表示为：

$$pre(c) \rightarrow post(c)[e/v]$$

我们现在说，给出一个流程图程序的极化节点，证明其前置条件和后置条件的一致性能保证程序的部分正确性。事实上，这保证了一个更强的属性：

每次执行，若其开始于满足程序的初始条件的状态，当程序的控制点在某位置时，那么关联在该位置的条件成立。

特别地，当到达了程序终止节点的入边时，关联在该位置的最终断言必须成立。上述属性的证明是基于在始于满足初始条件状态的执行序列的前缀长度上的简单归纳。归纳的奠基条件显然成立。归纳步骤解释如下。假设该前缀以某位置的状态 a 结尾，a 上条件为 $pre(c)$。然后 c 被执行了。在此情况下，根据归纳假设，$a \models_S pre(c)$。那么，上文中三种情况规定的前置条件和后置条件的对应关系保证了若 c 被执行，则我们到达了满足 $post(c)$ 的状态 b。状态 b 使原前缀的长度增加一，形成新的前缀，其中每个状态都满足流程图中对应的位置条件。

图 7.2 中的流程图程序计算了自然数 $x1$ 除以 $x2$ 的整数除法。结果在计算终结时存储在变量 $y1$ 中，余数存在 $y2$ 中。初始条件是 $x1 \geqslant 0 \wedge x2 > 0$。最终断言是 $(x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant 0 \wedge y2 < x2$。

找出证明所需的不变式可能是一项艰巨的工作。有多种启发算法和工具试图发现不变式[75]。然而，根据可计算性理论，我们知道不可能有一种全自动的发现不变式的方法，换言之，该问题是不可判定的。寻找不变式的第一步是仔细观察程序，研究其行为，例如，可模拟一些执行。然后，我们可以试图发现在每个位置的不同程序变量间的关系。

这个过程通常涉及一些试验和错误。我们从一组初步的位置不变式出发，继而尝试验证极

化节点的前置条件和后置条件间的一致性条件。假设 c 是一个肯定判定的极化节点，且我们不能证明 $pre(c) \wedge p \rightarrow post(c)$。之所以这样是因为 $pre(c)$过弱，没有包含在进入 c 位置时程序变量间的足够的关联信息。另一个可能性是 $post(c)$不是在离开 c 位置时的正确的不变式，当程序的控制点在该位置时，一些它规定的关联信息不成立。

示例：整数除法

该程序的输入是非负整数 $x1$ 和正整数 $x2$。程序是整数除法 $x2$ 除以 $x1$，结果在 $y1$ 中，余数在 $y2$ 中。因此，初始条件是 $x1 \geqslant 0 \wedge x2 > 0$。最终断言是 $(x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant 0 \wedge y2 < x2$。

图 7.2 中的位置用字母 $A \sim F$ 做了标记。我们首先将初始条件归于位置 A，最终断言归于位置 F。我们观察到如下事实：在开始时，$y2$ 得到 $x1$ 的值。在主循环中，只要 $y2$ 不小于 $x2$，我们就从 $y2$ 中减去 $x2$，并给 $y1$ 加 1。在位置 C，我们从 $y2$ 中减去 $x2$ 的次数是 $y1$。总的来说，我们共从 $y2$ 中减去了 $y1 \times x2$。如果将此乘积加上 $y2$ 中剩余的值，其和应是 $x1$ 的初始值（也是目前的值）。也就是说，在位置 C，我们有 $x1 \equiv y1 \times x2 + y2$。进一步地，很容易看出 $y2 \geqslant 0$ 在此处成立。使用相似的方法，我们设定如下不变式：

$$\varphi(A) = x1 \geqslant 0 \wedge x2 > 0$$
$$\varphi(B) = x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0$$
$$\varphi(C) = (x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant 0$$
$$\varphi(D) = (x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant x2$$
$$\varphi(E) = (x1 \equiv y1 \times x2 + y2 - x2) \wedge y2 - x2 \geqslant 0$$
$$\varphi(F) = (x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant 0 \wedge y2 < x2$$

为证明一致性，我们需要使用一些一阶逻辑证明系统来证明如下蕴含式：

$$\varphi(A) \rightarrow \varphi(B)[0/y1] =$$
$$(x1 \geqslant 0 \wedge x2 > 0) \rightarrow (x1 \geqslant 0 \wedge x2 > 0 \wedge 0 \equiv 0)$$
$$\varphi(B) \rightarrow \varphi(C)[x1/y2] =$$
$$(x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0) \rightarrow ((x1 \equiv y1 \times x2 + x1) \wedge x1 \geqslant 0)$$
$$(\varphi(C) \wedge y2 \geqslant x2) \rightarrow \varphi(D) =$$
$$((x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant 0 \wedge y2 \geqslant x2) \rightarrow ((x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant x2)$$
$$(\varphi(C) \wedge \neg y2 \geqslant x2) \rightarrow \varphi(F) =$$
$$((x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant 0 \wedge \neg y2 \geqslant x2) \rightarrow$$
$$((x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant 0 \wedge y2 < x2)$$
$$\varphi(D) \rightarrow \varphi(E)[y1 + 1/y1] =$$
$$((x1 \equiv y1 \times x2 + y2) \wedge y2 \geqslant x2) \rightarrow$$
$$((x1 \equiv (y1 + 1) \times x2 + y2 - x2) \wedge y2 - x2 \geqslant 0)$$
$$\varphi(E) \rightarrow \varphi(C)[y2 - x2/y2] =$$
$$((x1 \equiv y1 \times x2 + y2 - x2) \wedge y2 - x2 \geqslant 0) \rightarrow$$
$$((x1 \equiv y1 \times x2 + y2 - x2) \wedge y2 - x2 \geqslant 0)$$

练习 7.1.1　使用一个自动化的定理证明器来证明表明点 D 和 E 以及点 E 和 C 之间一致性的蕴含式。

7.2　含数组变量的验证

上一节所呈现的证明系统仅处理某域$\mathcal{D}$上的简单变量。当允许数组变量时，该证明系统需作

修改。本节为高级内容，可放心跳过。

7.2.1 含数组变量赋值的问题

为了解所涉及的问题，考虑赋值 $x[x[1]]:=2$。它将值 2 赋给 $x[x[1]]$，其中索引（index）$x[1]$ 根据赋值前的状态计算而得。然而，若在赋值前 $x[1]=1$，将 2 赋值给 $x[1]$ 就会有副作用（因为此时 $x[1]$ 和 $x[x[1]]$ 是同一内存位置的两个引用）。假设 c 的前置条件是 $pre(c)=x[1]\equiv 1$，后置条件是 $post(c)=x[x[1]]\equiv 2$。断言

$$\{x[1]\equiv 1 \wedge x[2]\equiv 3\}x[x[1]]:=2\{x[x[1]]\equiv 2\} \tag{7.1}$$

是不正确的。它表明在赋值后，我们有 $x[x[1]]\equiv 2$。然而，因为初始时 $x[1]\equiv 1$，我们现在有 $x[1]\equiv 2$，所以 $x[x[1]]$ 现在是 $x[2]$，其值应为 3 而非 2。

幼稚地应用上一节中的证明系统允许按如下方法“证明”不正确的断言（7.1）。让我们根据上节中的形式化方法来相对化 $post(c)$，使其对应赋值前的状态。我们可以得到

$$relpost(c) = x[x[1]]\equiv 2[2/x[x[1]]] = 2\equiv 2 = true \tag{7.2}$$

现在，使用式（7.2）我们得到 $pre(c)\rightarrow relpost(c)$ 是 $x[1]\equiv 1\rightarrow true$，其逻辑等价于 $true$。这确立了式（7.1）。我们的证明系统在这里允许我们证明不正确的事实，因而是无效的。所以，我们需要改变它。我们通过改变数组赋值的相对化来实现。

7.2.2 修改证明系统

为正确处理数组变量，我们需要扩展第 3 章中定义的一阶逻辑。语法实体项（在 3.2 节中定义）也将包含数组元素的引用。除此以外，一类新的项会被引入，它的行为类似于函数式编程语言中的 if-then-else 结构。项的扩展 BNF 定义是：

$$term ::= var \mid func(term,term,\cdots,term) \mid const \mid$$
$$if(form,term,term) \mid arr\_var[term] \mid (arr\_var,term:term)$$

结构 arr_var 是指数组变量的名称。我们假设程序中使用的数组变量集合与简单变量不相交。我们不允许在我们的一阶逻辑断言中量化数组变量。

赋值语句被扩展为可返回数组中元素的值。相应地，如果 x 是一个数组，i 是索引（我们这里忽略越界索引的难题），a 是赋值，那么 $a(\langle x, i\rangle)$ 给出了域$\mathcal{D}$中的一个值。解释函数 T_a 被扩展为如下形式：

$$T_a[x[e]] = a(\langle x, T_a[e]\rangle)$$
$$T_a[if(\varphi,e1,e2)] = \begin{cases} T_a[e1] & \text{如果 } M_a[\varphi] = \text{TRUE} \\ T_a[e2] & \text{否则} \end{cases}$$

if 结构没有为逻辑添加任何表达能力，因为它是可以被消除的。然而，若将其进行分配可能使公式大小有相当（指数级）的增长。考虑一个包含有 if 结构 $if(\varphi, e1, e2)$ 的公式 ψ，该公式可被转换成中间形式 η，其子公式 $if(\varphi, e1, e2)$ 被替换成一个新的变量 v。那么原始公式 ψ 等价于

$$(\varphi \wedge \eta[e1/v]) \vee (\neg\varphi \wedge \eta[e2/v]) \tag{7.3}$$

该过程一直重复直到所有的 if 结构都被消除（注意 if 子公式的多个相同出现应被同时消除）。

例如，考虑公式

$$x1 > if(x2 > 3, x4, if(x3 > x5, 2, 4)) \tag{7.4}$$

根据上面的转换，当用新变量 v 替换式（7.4）中最里面的 if 时，我们得到如下的公式：

$$\eta = x1 > if(x2 > 3, x4, v), \varphi = x3 > x5, e1 = 2, e2 = 4$$

根据式（7.3）我们得到等价于式（7.4）的如下公式：

$$(x3 > x5) \wedge (x1 > if(x2 > 3, x4, 2)) \vee$$
$$\neg(x3 > x5) \wedge (x1 > if(x2 > 3, x4, 4))$$

注意得到的公式现在包括两个 if 结构，需要将它们消除。

练习 7.2.1　消除上述公式中的所有 if 结构。

本节中项的扩展定义包含另一个结构：对于数组变量 x，让 $(x;e1 : e2)$ 成为 x 的变体（variant），其中 $e1$ 和 $e2$ 是表达式。非形式化地，变体 $(x;e1 : e2)$ 和 x 的不同仅在于它的 $e1$ 元素的值被设成 $e2$。我们在一阶公式中使用变体来代替数组变量（注意变体是一个数组，而不是数组元素）。其形式化的语义如下：

$$T_a[(x;e1 : e2)[e3]] = \begin{cases} T_a[e2] & \text{如果 } T_a[e1] = T_a[e3] \\ a((x, T_a[e3])) & \text{否则} \end{cases}$$

该变体符号也可使用如下转换从公式中消除：

$$(x;e1 : e2)[e3] = if(e1 \equiv e3, e2, x[e3]) \tag{7.5}$$

为了获得数组赋值的正确替换，用加撇号的版本来写赋值后的变量，然后用非加撇号的变量组成的表达式来代替。微妙之处在于我们需要处理数组变量的加撇号和非加撇号的版本，而不是数组元素的加撇号和非加撇号的版本。数组变量的定义可帮助我们建立此联系。

如果被赋值的变量是简单变量，那么替换方式就如上一节所述。然而，如果被赋值变量是数组变量，那么它必须被其变体替换。所以，如果赋值 c 是 $x[e1] := e2$，那么赋值后的 x' 与之前的 $(x;e1 : e2)$ 相同。正如在上一节中描述的简单赋值的情况，不需要首先用变量的加撇号版本替代变量，因为这样做仅仅是为了解释证明系统。在上述的赋值中，可以直接用变体 $(x;e1 : e2)$ 来替代 x。那么，相对化的替代就该是

$$relpost(c) = post(c)[(x;e1 : e2)/x]$$

这样，所有变体的实例都可以用转换规则式（7.5）消除。

我们现在再来看 c 是 $x[x[1]] := 2$，$post(c) = x[x[1]] \equiv 2$ 这个例子。相对化的后置条件 $relpost(c)$ 是

$$x[x[1]] \equiv 2[(x;x[1] : 2)/x]$$

执行 x 的变体替换，我们得到

$$(x;x[1] : 2)[(x;x[1] : 2)[1]] \equiv 2$$

现在，我们可以消除变体符号的第二次出现，并按上文说的用一个 if 结构来代替。我们得到：

$$(x;x[1] : 2)[if(x[1] \equiv 1, 2, x[1])] \equiv 2 \tag{7.6}$$

消除剩下的变体出现，我们有

$$if(x[1] \equiv \underline{if(x[1] \equiv 1, 2, x[1])}, 2, x[\underline{if(x[1] \equiv 1, 2, x[1])}]) \equiv 2 \tag{7.7}$$

我们对 if 结构的相同的两次出现加了下划线。所以，我们能够同时消除它们。用 v 代替下划线部分，我们得到 $\eta = if(x[1] \equiv v, 2, x[v]) \equiv 2$，$\varphi = x[1] \equiv 1$，$e1 = 2$，$e2 = x[1]$。那么，根据式（7.3）中的转换方法，我们得到 $(x[1] \equiv 1 \wedge if(x[1] \equiv 2, 2, x[2] \equiv 2) \vee (x[1] \not\equiv 1 \wedge if(x[1] \equiv x[1], 2, x[x[1]]) \equiv 2)$。

我们可以做一些逻辑简化。首先 $if(x[1] \equiv x[1], 2, x[x[1]]) \equiv 2$。进一步地，给出 $x[1] \equiv 1$，我们有 $if(x[1] \equiv 2, 2, x[2]) \equiv x[2]$。所以，我们得到简化了的公式

$$(x[1] \equiv 1 \wedge x[2] \equiv 2) \vee (x[1] \not\equiv 1) \tag{7.8}$$

通过对式（7.8）应用逻辑等价，我们最终得到

$$relpost(c) = x[1] \equiv 1 \rightarrow x[2] \equiv 2$$

显然，上述为得到相对化的后置条件的逻辑操作相当困难，而且难以理解。自动化的定理证明器可以帮助实施或是检查这些操作的正确性。

7.3 完全正确性

程序的*完全正确性*，记为 $\langle\varphi\rangle P\langle\psi\rangle$，规定如果程序 P 从满足初始条件 φ 的状态开始执行，那么它终止于满足公式 ψ 的状态。形式化地，对每一对程序状态 a，b，

如果 $a\models_S\varphi$，那么存在状态 b 使得 $P(a,b)$ 且 $b\models_S\psi$。

（该定义是基于流程图程序是*确定的*这一事实。）可以通过分别证明部分正确性和终止性来证明完全正确性（尽管我们稍后会阐述如何直接证明完全正确性）。通常，部分正确性成立的理由和终止性成立的理由会不同。所以，分开证明的方法，尽管不是必需的，但可能会有助于简化证明。终止性（再次说明是针对确定的程序）形式化的定义为，

如果 $a\models_S\varphi$，那么有状态 b 使得 $P(a,b)$。

在7.1节中介绍了一种使用在执行的有限前缀上做简单归纳的方法来证明部分正确性。该归纳阐明了如果相邻的前置条件和后置条件的一致性得到了证明，那么每一个始于满足初始条件状态的程序执行中的状态都满足关联在某程序位置上的不变式。特别地，如果程序终止，那么最后的状态必须满足关联在进入终止节点上的不变式，该不变式恰好就是程序的最终断言。

这种归纳推理不足以证明终止性，因为它处理的是任意长度的有限前缀。我们需要的是一个归纳原理，能考虑无限执行的可能性并能将其排除在外。因此我们在终止性证明中将使用另一种推理原则，该原则涉及良基域的概念。

一个*偏序域*（partially ordered domain）是一个二元对 $(W,\succ)$，其中 W 是集合，“$\succ$”是 W 上的偏序关系。回忆一下，对于一个偏序关系“$\succ$”，如下条件必须成立：

反自反性。没有 $u\in W$ 使得 $u\succ u$ 成立。

不对称性。对每个 u，$v\in W$，如果 $u\succ v$，那么 $v\succ u$ 不成立（我们也写作 $v\not\succ u$）。

传递性。如果对某 u，v，$w\in W$，$u\succ v$ 且 $v\succ w$，那么 $u\succ w$。

当 $u\succ v$ 或者 $u\equiv v$ 时我们记为 $u\succeq v$。当 $u\succeq v$ 时，我们也写作 $v\preceq u$。类似地，$u\succeq v$ 和 $v\preceq u$ 是一样的。

偏序域的例子有：

- 整数集合和通常意义下的“大于”关系“$>$”。
- 整数的有限集合的集合和集合包含关系“$\supset$”。
- 某有限字母表上的字符串集合，当 w 是 u 的子字符串有 $u\succ w$，即有字符串 v_1 和 v_2 使得 $u=v_1.w.v_2$。
- 某有限字母表（如拉丁字母表）上的字符串和字典序（即词在字典中出现的顺序）。
- 自然数的元组（如二元组或三元组）和元组间的字母序。和词典中的字母序类似，元组间的字母序给出了两个第一分量元素间的优先关系。如果它们不同，元组就按这些元素的关系排序。如果这些元素相同，优先级就依据第二分量，以此类推。例如对二元对，如果 $u_1\succ v_1$ 或 $u_1\equiv v_1$ 且 $u_2\succ v_2$，则 $(u_1,u_2)\succ(v_1,v_2)$。

良基域（well founded domain）是不包含无限递减序列的偏序域，换言之，不存在形如 $w_0\succ w_1\succ\cdots\succ w_n\succ\cdots$ 的序列。上文例子中第一个偏序域不是良基的。这是因为对于整数和通常意义的“大于”关系，有一个无限的递减序列，如 $0>-1>-2>\cdots$。如果用自然数代替整数，我们就获得了一个良基域。按字母序排序的字符串也不是良基的，例如考虑无限序列 $ab\succ aab\succ aaab\succ\cdots$。上文中其余的偏序域都是良基的。

我们需要按以下步骤证明终止性。

- 选择一个良基域（W，$\succ$）。要求 $W \subseteq \mathcal{D}$，且关系 $\succ$ 必须是用程序签名可表达的。
- 为流程图的每一个位置添加一个公式（不变式）和一个表达式。添加在开始节点出边上的公式是初始条件。（同时证明终止性和部分正确性是可能的，可通过在终止节点的入边上关联最终断言来实现。）在任意的程序执行中，当程序的控制点在关联有表达式 e 的某位置上，若此时状态为 a，则将 $T_a(e)$ 与该状态相关联。
- 仿照部分正确性的证明，证明每个极化节点的前置条件和后置条件间的一致性。
- 证明执行中和每个状态关联的值满足如下条件：
  - 和一个流程图位置关联的表达式 e 根据执行中的某个状态（当程序计数在该位置时）计算表达式值，其结果应在 W 集合中。
  - 在程序的每次执行中，当程序从一个位置运行到它的后继位置时，关联的表达式的值不增加。
  - 在程序的每次执行中，在遍历流程图的圈（循环）的过程中，存在某个点使得关联的表达式从一个位置到它的后继位置时其表达式值会减少。

令 c 为某个极化节点，$pre(c)$ 和 $post(c)$ 分别是它的前置条件和后置条件，e_1 和 e_2 分别是关联在入边和出边上的表达式。我们现在将形式化需要证明的条件。

首先，前置条件和后置条件间的一致性条件和部分正确性的证明是一样的。有三种情况：

- 肯定判定 p。那么

$$(pre(c) \wedge p) \rightarrow post(c) \tag{7.9}$$

- 否定判定 p。那么

$$(pre(c) \wedge \neg p) \rightarrow post(c) \tag{7.10}$$

- 赋值 $v := e$。那么

$$pre(c) \rightarrow post(c)[e/v] \tag{7.11}$$

这就使得每次执行到某个特定的位置时，关联在该位置上的不变式成立。

为证明关联在一个位置上的表达式 e 的值是在集合 W 中，我们需要证明

$$\varphi \rightarrow (e \in W) \tag{7.12}$$

回忆一下，$e \in W$ 通常不是一阶逻辑公式。然而，在这里，根据被验证的程序，它通常能被转换成一阶项。例如，当程序域是整数时，可能出现良基域是自然数和通常意义的“$>$”关系的情况。这时式（7.12）就变成了 $\varphi \rightarrow (e \geqslant 0)$。

对每个满足 $a \models_S pre(c)$ 的状态 a 和满足 $b \models_S post(c)$ 的后继状态 b，为证明 $T_a(e_1) \geqslant T_b(e_2)$，需要相对化表达式 e_1 和 e_2 中的一个，以使得两者皆针对同一状态。我们选择相对化 e_2 来得到 $rel_c(e_2)$。和 7.1 节中一样，我们能使用一组加撇号的变量来表示执行 c 后的程序变量的值。之后根据（极化）节点 c 的类型我们用非加撇号的变量组成的表达式代替加撇号的变量。

- 就肯定或否定判定 c 而言，对每个程序变量 u 我们有 $u \equiv u'$。因此 $rel_c(e_2) \equiv e_2$。
- 就赋值 $v := e$ 而言，对每个 $u \neq v$ 我们有 $u \equiv u'$，以及 $v' \equiv e$。因此结果为 $rel_{v:=e}(e_2) \equiv e_2[e/v]$。

为证明在执行极化节点 c 的过程中关联在各位置上的表达式 e 的值没有增加，需要证明蕴含式

$$pre(c) \rightarrow (e_1 \geq rel_c(e_2)) \tag{7.13}$$

为证明沿着流程图中的圈 e 的值会减小，需要证明至少存在一个极化节点 c 使得

$$pre(c) \rightarrow (e_1 \succ rel_c(e_2)) \tag{7.14}$$

现在我们可以论证程序必须终止。证明是基于矛盾论证。假设对程序 P 式（7.9）～（7.14）已被证明，且 P 不终止。考虑关联在 P 的某个非终止的执行上的值的序列。因为域（W，$\succ$）是良基的，所以不可能形成一个无限递减的序列。同时因为它是由式（7.13）构建的

一个非递减的序列，所以必然在某点该序列会达到一个常量值。考虑从该点开始的关联值保持为常量的执行后缀。因为程序只有有限多个位置，该后缀不可能经过无限多个节点，所以它必然经过某个圈。但是根据式（7.14），关联的值必然存在减小的情况，这就有了矛盾。

我们现在将证明图7.2中的流程图程序的终止性。我们选择良基域（Nat，$>$），即自然数和通常的排序关系。位置A到F的不变式和表达式的标注如下：

$$\varphi(A) = x1 \geqslant 0 \wedge x2 > 0 \quad e(A) = x1$$
$$\varphi(B) = x1 \geqslant 0 \wedge x2 > 0 \quad e(B) = x1$$
$$\varphi(C) = x2 > 0 \wedge y2 \geqslant 0 \quad e(C) = y2$$
$$\varphi(D) = x2 > 0 \wedge y2 \geqslant x2 \quad e(D) = y2$$
$$\varphi(E) = x2 > 0 \wedge y2 \geqslant x2 \quad e(E) = y2$$
$$\varphi(F) = y2 \geqslant 0 \quad e(F) = y2$$

为证明表达式返回了良基域中的值，我们立刻就有

$$\varphi(A) \rightarrow x1 \in Nat \quad \varphi(B) \rightarrow x1 \in Nat$$
$$\varphi(C) \rightarrow y2 \in Nat \quad \varphi(D) \rightarrow y2 \in Nat$$
$$\varphi(E) \rightarrow y2 \in Nat \quad \varphi(F) \rightarrow y2 \in Nat$$

为证明一致性，我们需要证明以下蕴含式：

$$\varphi(A) \rightarrow \varphi(B)[0/y1] =$$
$$(x1 \geqslant 0 \wedge x2 > 0) \rightarrow (x1 \geqslant 0 \wedge x2 > 0)$$
$$\varphi(B) \rightarrow \varphi(C)[x1/y2] =$$
$$(x1 \geqslant 0 \wedge x2 > 0) \rightarrow (x2 > 0 \wedge x1 \geqslant 0)$$
$$(\varphi(C) \wedge y2 \geqslant x2) \rightarrow \varphi(D) =$$
$$(x2 > 0 \wedge y2 \geqslant 0 \wedge y2 \geqslant x2) \rightarrow (x2 > 0 \wedge y2 \geqslant x2)$$
$$(\varphi(C) \wedge \neg y2 \geqslant x2) \rightarrow \varphi(F) =$$
$$(x2 > 0 \wedge y2 \geqslant 0 \wedge \neg y2 \geqslant x2) \rightarrow y2 \geqslant 0$$
$$\varphi(D) \rightarrow \varphi(E)[y1 + 1/y1] =$$
$$(x2 > 0 \wedge y2 \geqslant x2) \rightarrow (x2 > 0 \wedge y2 \geqslant x2)$$
$$\varphi(E) \rightarrow \varphi(C)[y2 - x2/y2] =$$
$$(x2 > 0 \wedge y2 \geqslant x2) \rightarrow (x2 > 0 \wedge y2 - x2 \geqslant 0)$$

为证明两个相继位置上关联的值没有增加，且程序控制点在循环$C \rightarrow D \rightarrow E \rightarrow C$上从$E$移动到$C$时关联值有一个适当的减少，我们有：

$$\varphi(A) \rightarrow e(A) \geq e(B)[0/y1] =$$
$$(x1 \geqslant 0 \wedge x2 > 0) \rightarrow x1 \geqslant x1$$
$$\varphi(B) \rightarrow e(B) \geq e(C)[x1/y2] =$$
$$(x1 \geqslant 0 \wedge x2 > 0) \rightarrow x1 \geqslant x1$$
$$(\varphi(C) \wedge y2 \geqslant x2) \rightarrow e(C) \geq e(D) =$$
$$(x2 > 0 \wedge y2 \geqslant 0 \wedge y2 \geqslant x2) \rightarrow y2 \geqslant y2$$
$$(\varphi(C) \wedge \neg y2 \geqslant x2) \rightarrow e(C) \geq e(F) =$$
$$(x2 > 0 \wedge y2 \geqslant 0 \wedge \neg y2 \geqslant x2) \rightarrow y2 \geqslant y2$$
$$\varphi(D) \rightarrow e(D) \geq e(E)[(y1 + 1)/y1] =$$
$$(x2 > 0 \wedge y2 \geqslant x2) \rightarrow y2 \geqslant y2$$
$$\varphi(E) \rightarrow e(E) > e(C)[(y2 - x2)/y2] =$$
$$(x2 > 0 \wedge y2 \geqslant x2 \wedge x2 > 0) \rightarrow y2 > y2 - x2$$

7.4　公理式程序验证

验证流程图程序有个缺点，必须首先将程序从文本代码转换成流程图。现在流程图用的已经不多了，或许是因为它难以用在大型程序上。Hoare [63] 开发了一个证明系统，同时包括逻辑和代码片段。该证明系统允许我们直接验证程序代码。而且，它促进了组合验证，允许我们分别证明程序中不同的顺序部分，稍后再将证明组合在一起。

该逻辑构建筑在某些一阶演绎系统之上。除了一阶断言外，该逻辑允许 $\{\varphi\}S\{\psi\}$ 形式的断言，其中 S 是程序的一个部分，φ 和 ψ 是一阶公式。这些断言被称做 Hoare 三元组。所使用的程序（和程序片段）的语法用 BNF 给出：

$$S ::= v := e \mid skip \mid S;S \mid$$
$$if\ p\ then\ S\ else\ S\ fi \mid while\ p\ do\ S$$

其中 v 是变量，e 是一阶表达式，p 是非量化的一阶断言。Hoare 三元组 $\{\varphi\}S\{\psi\}$ 的含义如下：如果 S 的执行始于满足 φ 的状态，且从该状态起 S 能够终止，那么它能到达满足 ψ 的状态。显然，如果 S 是整个程序，那么 $\{\varphi\}S\{\psi\}$ 就是其部分正确性的声明，它的初始条件为 φ，最终断言为 ψ。

注意允许的程序不包含任何 goto 语句。因此，使用上述语法形成的程序 S 的任何部分都仅能从 S 的开头开始执行，如果其能终止，则结束于 S 的结尾。该假设对下面证明规则的公式化至关重要。Hoare 逻辑对 goto 语句的处理参见 [12]。Hoare 逻辑的公理和证明规则在下面给出。

7.4.1　赋值公理

该公理使人想起 7.1 节中 Floyd 证明系统下证明赋值的前置条件和后置条件一致性的方法。它取描述赋值 $v := e$ 执行后的状态的后置条件 φ，然后转换成用赋值前的变量描述的对应的前置条件。

$$\{\varphi[e/v]\}v := e\{\varphi\}.$$

如果涉及数组变量，那么需要采取与 7.2 节中讨论的内容的相同的方法。

7.4.2　空语句公理

$$\{\varphi\}skip\{\varphi\}$$

执行一个空语句指令不改变任何程序变量的值。这反映在公理中，该公理取相同的公式作为前置条件和后置条件。

7.4.3　左强化规则

该规则用于加强前置条件，即 $\{\varphi'\}S\{\psi\}$ 已被证明，想要将前置条件 φ' 强化为 φ（回忆一下，这意味着 $\varphi \rightarrow \varphi'$）。

$$\frac{\varphi \rightarrow \varphi', \{\varphi'\}S\{\psi\}}{\{\varphi\}S\{\psi\}}$$

为证明该规则，考虑任意满足 φ 的状态。然后根据蕴含式，它也满足 φ'。但是根据 $\{\varphi'\}S\{\psi\}$，如果 S 始于该状态并且会终止，那么 ψ 在它完成时成立。因此 $\{\varphi\}S\{\psi\}$。

本规则经常和赋值公理一起使用。如果我们想要证明 $\{\varphi\}v := e\{\psi\}$，我们通常首先获得弱于 φ 的前置条件 $\varphi[e/v]$，然后使用左强化规则证明 $\varphi \rightarrow \psi[e/v]$ 来强化前置条件。回忆一下，在 Floyd 证明系统中这两个证明步骤是同时执行的。

衍生证明规则（derived proof rule）经常被加入到 Hoare 证明系统中。这些额外的证明规则

将几个另外的证明规则和公理合并成一条规则，使更简单（更短）的证明成为可能。例如，可以合并赋值公理和左强化规则，得到如下衍生规则：

$$\frac{\varphi \rightarrow \psi[e/v], \{\psi[e/v]\}v := e\{\psi\}}{\{\varphi\}v := e\{\psi\}}$$

7.4.4 右弱化规则

本规则用于弱化后置条件，即 $\{\varphi\}S\{\psi'\}$ 已被证明，我们想要将后置条件 ψ' 弱化为 ψ（回忆一下，这意味着 $\psi' \rightarrow \psi$）。因为 $\{\varphi\}S\{\psi'\}$，如果 S 始于满足 φ 的状态且会终止，那么它完成时 ψ' 成立。但根据蕴含式，任何满足 ψ' 的状态也满足 ψ。因此 $\{\varphi\}S\{\psi\}$。

$$\frac{\{\varphi\}S\{\psi'\}, \psi \rightarrow \psi'}{\{\varphi\}S\{\psi\}}$$

注意在 Hoare 证明系统中，可以强化前置条件和弱化后置条件。

7.4.5 顺序组合规则

本规则允许我们对顺序组合证明部分正确性，即在分别对组件 S_1 和 S_2 证明部分正确性后得到 $\{\varphi\}S_1;S_2\{\psi\}$。这通过使用一个同时为 S_1 的后置条件和 S_2 的前置条件的断言 η 来达到。

$$\frac{\{\varphi\}S_1\{\eta\}, \{\eta\}S_2\{\psi\}}{\{\varphi\}S_1;S_2\{\psi\}}$$

通过首先分别验证代码的组件来验证代码是一项有用的策略，称为组合性（compositionality）。这是针对顺序组合规则的情况，因为 S_1 和 S_2 的证明是分别进行的，然后对 $S_1;S_2$ 的证明再由这些证明合并而得。

假设 S_1 的后置条件和 S_2 的前置条件不相同，例如，$\{\varphi\}S_1\{\eta_1\}$ 和 $\{\eta_2\}S_2\{\psi\}$ 都已被证明。当 $\eta_1 \rightarrow \eta_2$ 时仍然可以证明 $\{\varphi\}S_1;S_2\{\psi\}$。实现的方法可以是使用右弱化规则来弱化 S_1 的后置条件，使其从 η_1 弱化为 η_2，或是用左强化规则来强化 S_2 的前置条件，使其从从 η_2 强化为 η_1。我们也可以导出一条额外的证明规则，它的作用是：

$$\frac{\{\varphi\}S_1\{\eta_1\}, \eta_1 \rightarrow \eta_2, \{\eta_2\}S_2\{\psi\}}{\{\varphi\}S_1;S_2\{\psi\}}$$

7.4.6 if-then-else 规则

为证明 $\{\varphi\}if\ p\ then\ S_1\ else\ S_2\ fi\{\psi\}$，我们考虑两种情况：

- p 成立。因此 S_1 执行前 $\varphi \wedge p$ 成立，这可被用作一个前置条件。
- p 不成立。因此 S_2 执行前 $\varphi \wedge \neg p$ 成立，这可被用作一个前置条件。

该规则的前提是当 if-then-else 结构结束时 ψ 在两种情况下都成立。

$$\frac{\{\varphi \wedge p\}S_1\{\psi\}, \{\varphi \wedge \neg p\}S_2\{\psi\}}{\{\varphi\}if\ p\ then\ S_1\ else\ S_2\ fi\{\psi\}}$$

7.4.7 while 规则

while 规则使用一个不变式 φ。该不变式需要在循环执行前以及每次迭代末尾都成立。在执行任何循环体的迭代前，我们也知道循环条件 p 成立，因为否则我们会是退出循环而不是再次执行循环体。因此，我们用 $\varphi \wedge p$ 作为循环体 S 的前置条件。我们通过证明 φ 在每次循环体执行后（从任意满足 $\varphi \wedge p$ 的语句开始）都成立来证明它是不变式。所以，当循环结束时，φ 也必须成立。进一步地，我们也知道结束时循环条件 p 不成立，因而循环的后置条件是 $\varphi \wedge \neg p$。

$$\frac{\{\varphi \wedge p\}S\{\varphi\}}{\{\varphi\}\textit{while } p \textit{ do } S \textit{ end}\{\varphi \wedge \neg p\}}$$

7.4.8　begin-end 规则

本证明规则反映了一个事实，begin 和 end 语句的行为就像算术中的括号一样将各结构结合在一起，且没有副作用。

$$\frac{\{\varphi\}S\{\psi\}}{\{\varphi\}\textit{begin } S \textit{ end}\{\psi\}}$$

练习 7.4.1　考虑下面的 Hoare 风格的证明规则。它们中有些是不正确的，可用于证明程序不正确的属性。找出下面不正确的证明规则并给出理由。

$$\frac{\{\varphi'\}S\{\psi\},\varphi' \rightarrow \varphi}{\{\varphi\}S\{\psi\}}$$

$$\frac{\{\varphi\}S_1\{\textit{false}\}}{\{\varphi\}S_1;S_2\{\psi\}}$$

$$\frac{\{\varphi \wedge p\}S\{\psi_1\},\{\varphi \wedge \neg p\}S\{\psi_2\}}{\{\varphi\}S\{\psi_1 \vee \psi_2\}}$$

$$\frac{\{\varphi\}S\{\varphi\}}{\{\varphi\}\textit{while } p \textit{ do } S \textit{ end}\{\varphi \wedge \neg p\}}$$

$$\frac{\{\varphi\}S\{\varphi\}}{\{\varphi\}\textit{while } p \textit{ do } S \textit{ end}\{\varphi\}}$$

$$\frac{\{\varphi \wedge p\}S\{\varphi\},(\varphi \wedge \neg p) \rightarrow \psi}{\{\varphi\}\textit{while } p \textit{ do } S \textit{ end}\{\psi\}}$$

7.4.9　示例：整数除法

下面的程序计算 $x1$ 除以 $x2$ 的整数除法。这是 7.1 节中验证的程序的文本版本。

```
y1 := 0;
y2 := x1;
while y2>=x2 do
    y1 := y1+1;
    y2 := y2-x2;
end
```

首先，我们用断言注解程序，使得每个语句都有一个前置条件和后置条件。这些条件描述了程序中每个控制位置上变量之间的关系。正如意料之中的，这些断言和我们用于验证图 7.2 中流程图程序部分正确性的断言是一样的。

```
{x1≥0 ∧ x2>0}
y1 := 0;
{x1≥0 ∧ x2>0 ∧ y1≡0}
y2 := x1;
{x1≡y1×x2+y2 ∧ y2≥0}
while y2≥x1 do
    {x1≡y1×x2+y2 ∧ y2≥x2}
    y1 := y1+1;
```

$\{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}$

$y2 := y2 - x2$

$\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}$

end

$\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0 \wedge y2 < x2\}$

令 S_1 代表开始两个赋值语句，即 $y1 := 0$；$y2 := x1$。S_2 代表 while 循环，即 *while* $y2 \geqslant x2$ *do* $y1 := y1 + 1$；$y2 := y2 - x2$；*end*。S_3 代表循环 S_2 内的两个赋值语句，即 $y1 := y1 + 1$；$y2 := y2 - x2$。

根据如图 7.3 所示的证明树，证明将由后向推理完成。从一个节点到它的直接后继的边表示该节点使用后继节点中证明的结果作为它的前提。我们在这里不说明自然数上的一阶逻辑蕴含式的证明有多简单。我们假设它们已用某些恰当的一阶证明系统证明了。我们将列出系统中出现的不同的待证明的目标。我们给每个目标一个数字，后面跟着用到的证明规则或公理的名字。

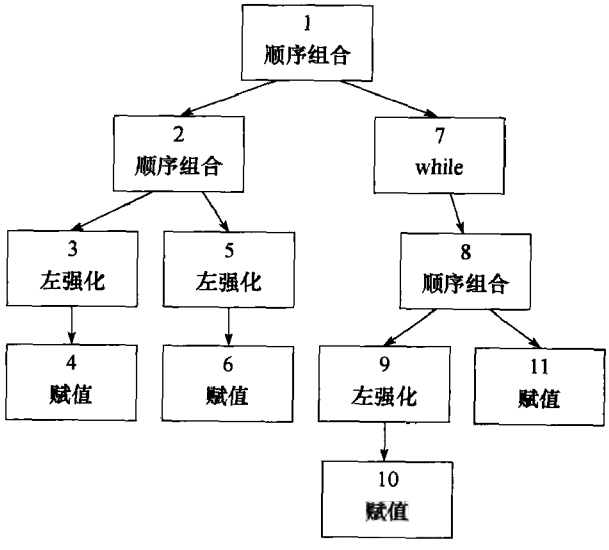


图 7.3 证明树

目标 1. 顺序组合

$$\frac{\{x1 \geqslant 0 \wedge x2 > 0\}S_1\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\} \quad \{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}S_2 \quad \{x1 \equiv y1 \times x2 + y2 \wedge x2 > y2 \wedge y2 \geqslant 0 \wedge y2 < x2\}}{\{x1 \geqslant 0 \wedge x2 > 0\}S_1;S_2} \quad \{x1 \equiv y1 \times x2 + y2 \wedge x2 > y2 \wedge y2 \geqslant 0 \wedge y2 < x2\}$$

目标 2. 顺序组合

$$\frac{\{x1 \geqslant 0 \wedge x2 > 0\}y1 := 0\{x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0\} \quad \{x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0\}y2 := x1\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}}{\{x1 \geqslant 0 \wedge x2 > 0\}S_1\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}}$$

目标 3. 左强化

$$\frac{(x1 \geqslant 0 \wedge x2 > 0) \rightarrow (x1 \geqslant 0 \wedge x2 > 0 \wedge 0 \equiv 0) \quad \{x1 \geqslant 0 \wedge x2 > 0 \wedge 0 \equiv 0\}y1 := 0\{x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0\}}{\{x1 \geqslant 0 \wedge x2 > 0\}y1 := 0\{x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0\}}$$

目标 4. 赋值

$$\{x1 \geqslant 0 \wedge x2 > 0 \wedge 0 \equiv 0\}y1 := 0\{x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0\}$$

目标 5. 左强化

$$(x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0) \rightarrow (x1 \equiv y1 \times x2 + x1 \wedge x1 \geqslant 0)$$

$$\frac{\{x1 \equiv y1 \times x2 + x1 \wedge x1 \geqslant 0\}y2 := x1\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}}{\{x1 \geqslant 0 \wedge x2 > 0 \wedge y1 \equiv 0\}y2 := x1\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}}$$

目标 6. 赋值

$$\{x1 \equiv y1 \times x2 + x1 \wedge x1 \geqslant 0\}y2 := x1\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}$$

目标 7. while

$$\frac{\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0 \wedge y2 \geqslant x2\}S_3\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}}{\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}S_2\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0 \wedge y2 < x2\}}$$

注意我们发现这里当 $y2<x2$ 时 $\neg y2 \geqslant x2$。

目标 8. 顺序组合

$$\frac{\begin{array}{l}\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant x2\}y1 := y1 + 1 \\ \quad \{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\} \\ \{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}y2 := y2 - x2 \\ \quad \{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}\end{array}}{\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant x2\}S_3\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant 0\}}$$

目标 9. 左强化

$$\frac{\begin{array}{l}(x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant x2) \rightarrow \\ \quad (x1 \equiv (y1 + 1) \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0) \\ \{x1 \equiv (y1 + 1) \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}y1 := y1 + 1 \\ \quad \{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}\end{array}}{\begin{array}{l}\{x1 \equiv y1 \times x2 + y2 \wedge y2 \geqslant x2\}y1 := y1 + 1 \\ \quad \{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}\end{array}}$$

目标 10. 赋值

$$\{x1 \equiv (y1 + 1) \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}y1 := y1 + 1$$
$$\{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}$$

目标 11. 赋值

$$\{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 - x2 \geqslant 0\}y2 := y2 - x2$$
$$\{x1 \equiv y1 \times x2 + y2 - x2 \wedge y2 \geqslant 0\}$$

有几种证明系统扩展了 Hoare 证明系统用于验证并发程序 [11，48，126]。这些证明系统提供了处理共享变量、同步和异步通信以及过程调用的公理。它们通常为某一特定的编程语言加以调整，如并发 Pascal 或 CSP。感兴趣的读者可以参考列在本章末尾的书籍。

7.5　并发程序的验证

Manna 和 Pnueli [91] 提出了一个用于处理并发的通用的证明系统。该系统不直接和任何特定的语法挂钩，而是假设所考虑的程序被翻译成一组原子转换（参见 4.4 节）。这样就能提供一个统一的验证系统用于不同的编程语言和并发范型。证明规则也允许验证不同的时序属性，而不仅是部分和完全正确性。

蕴含（entailment）是一个形如 $\varphi \Rightarrow \psi$ 的公式，这是 $\square(\varphi \rightarrow \psi)$ 的速记法。符号 $\{\varphi\}t\{\psi\}$ 是在一些证明规则中作为前提出现的断言。它的意思是如果 t 从满足 φ 的状态开始执行，那么就会到达满足 ψ 的状态。这种符号表示显示了该前提以类似于 Hoare 三元组的方式运作。考虑一个转换 $t : en_t \rightarrow (v_1, \cdots, v_n) := (e_1, \cdots, e_n)$。它先通过一个判定谓词，接着执行一个多赋值。因此，

形如 $\{\varphi\}t\{\psi\}$ 的前提可被如下蕴含替换：

$$(\varphi \wedge p) \rightarrow (\psi[e_1/v_1,\cdots,e_n/v_n])$$

我们将以 Manna-Pnueli 风格介绍一组证明规则。

FOL：和在 Hoare 证明系统中一样，Manna 和 Pnueli 证明系统包含了一个在适于被验证程序的结构$\mathcal{S}$上的一些重要一阶逻辑的正确、完整的一阶公理化。每个在结构$\mathcal{S}$上成立的一阶公式 φ，即$\models_{\mathcal{S}}\varphi$，在任何问题执行中的任何地方都成立。因此它也是（任何程序的）不变式。

$$\frac{\models_{\mathcal{S}}\varphi}{\Box\varphi}$$

该规则能将$\mathcal{S}$中形如 $\varphi\rightarrow\psi$ 的公式提升为蕴含式，即$\Box(\varphi\rightarrow\psi)$。

INV：该规则用于证明一个断言 η 是不变式。回忆一下，Θ 表示程序的初始条件。

I1 $\Theta\rightarrow\varphi$

I2 对于每个 $t\in T,\{\varphi\}t\{\varphi\}$

I3 $\dfrac{\varphi\rightarrow\eta}{\Box\eta}$

规则 **INV** 通过归纳法证明了 η 是不变式：

I1 这是归纳的基础。它声明了 φ 在初始时成立。

I2 这是步骤。它声明了如果 φ 在执行过程中的某个状态成立，那么无论取哪个转换，它在下一状态依然成立。

I3 本步骤证明了只要 φ 成立，η 就成立。

证明规则 **INV** 一眼看去可能显得过于复杂；好像在 **I1** 和 **I2** 中出现的公式 φ 能用 η 本身代替。那么 **I3** 就显得多余了。这是不对的，理由是并非每个不变式都能用这种替代的方式证明，我们可能需要把 η 加强成 φ。考虑简化的证明规则 **INV′**：

I1′ $\Theta\rightarrow\eta$

I2′ $\dfrac{\text{对于每个 } t\in T,\{\eta\}t\{\eta\}}{\Box\eta}$

为了展示同时使用 **INV′**和 **FOL** 来证明必要的一阶蕴含式的方法可能不足以证明不变式，考虑一个自然数上的小（顺序）程序：

$$\begin{array}{l}\text{初始 } x=0,y=0;\\ m1:x:=x+1;\\ m2:y:=y+1;\text{goto } m1;\end{array}$$

对于该程序，初始条件是 Θ：$x\equiv 0\wedge y\equiv 0\wedge pc\equiv m1$。两个转换是：

$$t1:pc\equiv m1\rightarrow(x,pc):=(x+1,m2)$$

$$t2:pc\equiv m2\rightarrow(y,pc):=(y+1,m1)$$

考虑不变式 $\eta=pc\equiv m1\rightarrow x\equiv y$。这的确是程序的一个不变式。然而，它仅描述了程序计数器在 $m1$ 时的状态，没有给出其他状态的信息。我们马上会看到，这个不变式因为过弱而不能直接用于 **INV′**。考虑 $\{\eta\}t2\{\eta\}$。这变成了

$$((pc\equiv m1\rightarrow x\equiv y)\wedge pc\equiv m2)\rightarrow(m1\equiv m1\rightarrow x\equiv y+1)$$

这可简化成

$$((pc\equiv m1\rightarrow x\equiv y)\wedge pc\equiv m2)\rightarrow(x\equiv y+1)$$

该一阶逻辑断言可能不是总能成立，例如，考虑赋值 $\{pc\mapsto m2,\ x\mapsto 3,\ y\mapsto 3\}$。因此我们不能用 **FOL** 证明前提 **I2′**。不变式 η 没有提供足够的信息用于归纳。我们说 η 是*不可归纳的*。为证明 η，我们需要强化它。例如，我们能在 **INV** 中使用

$$\varphi = (pc \equiv m1 \rightarrow x \equiv y) \wedge (pc \equiv m2 \rightarrow x \equiv y+1)$$

证明规则 INV′和 FOL 能够一起用于证明一个程序的不变式。这些规则是正确的。问题是它们没有为不变式提供一个完整的规则集合。

练习 7.5.1　使用 INV 证明上面例子中的$\Box\eta$。

NEXT：用于证明形如 $\varphi \Rightarrow \bigcirc\psi$ 的属性的规则。

$$\frac{\begin{array}{ll}\text{I1} & \varphi \Rightarrow \bigvee_{t\in T} en_t \\ \text{I2} & \text{对于每个 } t \in T, \{\varphi\}t\{\psi\}\end{array}}{\varphi \Rightarrow \bigcirc\psi}$$

前提 I1 声明了在每个满足 φ 的状态中至少有一个可执行的转换。前提 I2 说明了每个始于 φ 的可执行的转换，它从满足 φ 的一个状态开始执行，结束于一个满足 ψ 的状态。因此 $\varphi \Rightarrow \bigcirc\psi$。

FCS：规则 FCS 用于将满足断言 φ 的状态集合划分成两个集合。一个集合包含满足 φ_1 的状态，另一个集合包含满足 φ_2 的状态。然后，证明 $\varphi_i \Rightarrow \bigcirc\psi$ 是通过分别对满足 φ_1 的状态和满足 φ_2 的状态证明来完成的。为获得该规则的一般化效果以使其能用于任意数目的 φ_i，此规则可以重复使用。

$$\textbf{FCS}\quad \frac{\begin{array}{l}\varphi \Rightarrow (\varphi_1 \vee \varphi_2) \\ \varphi_1 \Rightarrow \bigcirc\psi \\ \varphi_2 \Rightarrow \bigcirc\psi\end{array}}{\varphi \Rightarrow \bigcirc\psi}$$

FIMM：本规则将 $\varphi \Rightarrow \psi$ 看做 $\varphi \Rightarrow \Diamond\psi$ 的一个特例。

$$\frac{\varphi \Rightarrow \psi}{\varphi \Rightarrow \Diamond\psi}$$

FTRN：规则 FTRN 可以证明 $\varphi \Rightarrow \Diamond\psi$。首先找到某个被 φ 和 ψ 间的中间状态满足的断言 ν，然后通过证明 $\varphi \Rightarrow \Diamond\nu$ 和 $\nu \Rightarrow \Diamond\psi$ 来完成。

$$\frac{\begin{array}{l}\varphi \Rightarrow \Diamond\nu \\ \nu \Rightarrow \Diamond\psi\end{array}}{\varphi \Rightarrow \Diamond\psi}$$

FPRV：该规则将 $\varphi \Rightarrow \bigcirc\psi$ 看做 $\varphi \Rightarrow \Diamond\psi$ 的一个特例。

$$\textbf{FPRV}\quad \frac{\varphi \Rightarrow \bigcirc\psi}{\varphi \Rightarrow \Diamond\psi}$$

FSPLIT：规则 FSPLIT 允许分解 $\varphi \Rightarrow \Diamond\psi$ 的证明，方法是将满足 φ 的状态集合分解成满足 φ_1 的状态和满足 φ_2 的状态。然后只需要分别证明 $\varphi_i \Rightarrow \Diamond\psi$，$i=1$，2。

$$\textbf{FSPLIT}\quad \frac{\begin{array}{l}\varphi \Rightarrow (\varphi_1 \vee \varphi_2) \\ \varphi_1 \Rightarrow \Diamond\psi \\ \varphi_2 \Rightarrow \Diamond\psi\end{array}}{\varphi \Rightarrow \Diamond\psi}$$

RESP：令 $(W, \succ)$ 为一良基域，e 为一涉及程序中变量的表达式。

$$\frac{\begin{array}{ll}\text{R1} & \varphi \Rightarrow (\psi \vee \eta) \\ \text{R2} & \eta \Rightarrow e \in W \\ \text{R3} & (\eta \wedge (e \equiv v)) \Rightarrow \Diamond(\psi \vee (\eta \wedge e \prec v))\end{array}}{\varphi \Rightarrow \Diamond\psi}$$

其中 v 是新的变量，它不出现在程序中，或是在任何一个公式 φ，ψ 和 η 中，或是在表达式 e 中。

前提 R1 声明了在 φ 成立的每个状态中，ψ 或是 η 成立。前提 R2 声明了当 η 成立时，表达式

e 的值在集合 W 中。前提 **R3** 声明了如果 η 成立，且表达式 e 有某个值 v，那么最终要么 ψ 成立，要么 η 成立且 e 的值最终会减小到小于 v 的某个值。因此，根据 **R1**，如果 φ 成立但是 ψ 未成立，则 η 必须成立。根据 **R3**，要么 ψ 最终会成立，要么有另一个未来的状态使得表达式 e 计算得到一个更小的值。但是 ψ 永不发生是不可能的，否则会有一个满足 η 的无限状态序列，和一个 e 值无限递减的序列。这和 $(W, >)$ 是良基域这一事实相矛盾。因此，良基域是用来展示从 φ 成立的状态发展到 ψ 成立的状态的过程的。该发展过程用表达式 e 的值的减小来衡量。因为这种减小不能永远继续，所以 ψ 最终会成立。(注意该证明规则和 7.3 节中顺序程序的完全正确性的证明系统的相似性。)

还有许多其他能够使用的证明规则。证明公平性下可能发生的事件的证明规则在某种程度上更为复杂，这可以在 [92] 中找到。

最后，回忆 6.12 节中介绍的安全属性。在那一节中为该特殊类别的属性提供了一个简单的模型检验算法。安全属性在演绎验证环境中能更为简单地得到处理。

众多定义安全属性的方法中的一种是：为程序添加一个新的叫做*历史变量*的变量集合 H，并且允许将程序的相同或是扩展签名上的任意的程序表达式赋值给这些变量。(一个技术性的备注：为了完备性我们可能需要用这些变量来编码某些值的序列，可以通过添加能够存储值序列的变量，或是通过一个允许编码序列的域如自然数来实现。) 这些赋值能够保存程序历史的一些信息，因此以一种类似 6.12 节中描述的安全自动机的状态的方式来工作。那么一个安全属性就是任意使用程序和历史变量的不变式 φ，换言之，就是一个形如 $\Box\varphi$ 的 LTL 属性。对于这种情况，我们为证明安全属性所需要的时序证明规则只是 FOL 和 INV。

关于这种验证风格的更多内容，包括许多例子，可参见 [91, 92, 93]。

练习 7.5.2 对规则 **RESP** 的进一步研究表明该规则的结果可变成一个*直到*（“$\mathcal{U}$”）属性。然而，诸如 **FCS**、**FRTN**、**FPRV** 等规则也相应地需要调整。给出这些恰当的证明规则。

练习 7.5.3 为 4.6 节中的互斥程序证明互斥属性 $\Box\neg(pc_1 \equiv m5 \wedge pc_2 \equiv n5)$。

7.6 演绎验证的优点

演绎验证是证明正确性的一种综合性方法。它不局限于有限状态系统（不像其他模型检验技术）。它能够处理不同域（整数，实数）上的程序以及数据结构（栈，队列，树）。它甚至允许参数化程序的验证，例如有任意数目相同进程的程序。自动验证技术通常只能检验这类程序的特定实例，例如检验全部含有 1～7 个进程的实例。不可判定的结果表明了通常我们不能自动化参数化程序的验证 [10]，除此以外模型检验还易于发生状态空间爆炸，这很大程度上限制了能被检验的并发进程的数目。

演绎验证的过程通常涉及大量脑力劳动。它更多地由代码开发者以外的人员来完成。这有助于增加试图理解程序算法背后的直观内容的人数，因此能增加发现错误的几率。另一方面，演绎验证也涉及引入严密性以及使用数学理论，这有时候能带来一些回报，诸如：

- 发现代码中的错误并且纠正它们。
- 一般化被验证的算法以发现预料之外的情况。
- 更好地理解被验证的算法。

通常，模型检验提供了程序正确性不能保持情况下的有用证据，而演绎验证则提供了为什么程序能运行的直观解释。因此，如果可能的话，同时使用模型检验和定理证明能够提供任一情况下的信息。(有关模型检验不能找到一个反例时构建证明的自动生成的方法，参见 [117]。)

验证理论引入了多种有用的工具和概念来增加被开发代码的可靠性，甚至是在这些验证的

形式化方法还没有被使用的时候。一个例子是不变式的概念，即需要在整个代码执行过程中成立的断言。不变式通常将不同变量的值互相关联起来，以及将这些变量的值关联到代码中特定的位置上。一种增加代码可靠性的方法是要求程序员提供不变式作为代码开发过程的一部分。不变式能够用于*运行时验证*（runtime verification），方法是把不变式插入到代码中作为额外的谓词；如果某个应当成立的不变式未能成立，额外的代码会触发一个中断来禁止程序继续运行，并且立即报告这一问题。

当验证实际的代码较为困难的时候，验证一个抽象的、较少细节的版本至少能增加一点可靠性。这和盛行的自动验证的方法并无太大差别，后者通常在实际的验证过程前对实际代码进行抽象。尽管程序的验证可能是困难的，但验证程序背后的算法也许是可以接受的一项任务，并且它能够为代码提供更好的可信度。对一些至关重要的系统，故障可以引发巨大的损害，因此尽管完全的验证会耗费特别的人力和资源，也应该使用。

演绎式形式化验证中的概念可以在软件可靠性方法学中非形式化地使用，如10.3节中介绍的净室方法。最后，演绎验证证明系统也可以被看做是编程语言结构的形式化语义的数学定义。7.4节中的Hoare证明系统是一个经典例子，它能作为类Pascal编程语言的形式化语义定义。这有助于开发更简洁、更易于理解的语言。原则上，如果一个新的编程结构难以结合到证明系统中，这可能意味着它没有被完整地理解或是很好地定义。

7.7 演绎验证的缺点

演绎验证的一个问题是它非常耗时。在大型项目中，演绎验证可能成为项目的瓶颈。验证的速度明显慢于标准的编程速度。比起其他的形式化方法技术如测试和模型检验，演绎验证也是一个相当慢的方法。

我们通常是验证一个可能和实际程序不同的代码的简化模型。即使我们试图验证实际代码，证明系统也可能会为被验证的代码假设不同的语义解释。例如，证明系统为简单赋值 $x := x+1$ 给出的语义可能是增加 x。而实际的代码是对一个32位、补码格式的寄存器加1，并且当溢出发生时升高一个标志位（或是引发一个中断）。

错误可能因为其他原因而被植入到证明中。例如，这可能是因为一个错误的编译器（在罕见的情况下，甚至是因为错误的硬件）导致的结果。想要验证代码执行所依赖的所有层次基本是不可能的。然而，在某些极少的情况下是可以得到这样一个经过验证的层次体系的（硬件，编译器，代码）[19]。

计算机程序的发展增加了使用诸如过程调用和面向对象代码等结构。不幸的是，演绎验证未能表现出与之对应的扩展能力。并发程序比起顺序代码更易于出错，因此更需要验证。这类程序的验证特别困难。本质上，证明器经常必须同时使用所有相关进程的全局性知识。或许解决扩展性问题的方法不仅仅是试图开发更好的证明系统和自动化的证明器，还需要重视验证需求的更好的编程语言和编程习惯。在这个方向上已经迈出的一步是不变式的概念，不变式已经从程序验证领域扩散到程序开发社区。

演绎验证大部分是手动完成的，且极大地依赖于验证人员的智慧。验证人员可能会得到程序员或设计人员的帮助，他们需要提供恰当的不变式（在执行中的任何位置都成立的断言），以及中间断言（在执行过程中的某些点上成立的断言）。通常，自动获取这些断言是不可行的，尽管有一些启发式技巧。此外，部分证明中涉及简单逻辑蕴含式的内容无法自动化，这是因为证明系统中存在已经用数学方法证实了的局限。演绎验证需要大量的专门知识以及数学背景，要找到具有合适知识的人来进行验证可能会有难度。

手工建立的证明有可能会包含错误。由于整个验证过程的困难性以及复杂的本质，验证人

员也许会想使用“捷径”，简单的看上去毫无问题的蕴含式（例如 $x \geqslant y \rightarrow x > y$）或是简单的代码片段会被假设为正确的而没有去证明。由工具支持的演绎验证仍然有潜在的问题，那就是工具里的错误。当考虑验证人员完成工作的质量问题，或是验证工具的正确性问题，都会提出一个疑问：“谁来检验验证人员?”

最后，演绎验证中发生的一个常见错误是对被验证域做了过多的假设。验证人员在验证过程中会倾向于添加更多的看上去很简单而且有助于缩短证明的假设，然后错误地把这些假设当成公理。这些添加的假设可能会限制证明的一般性，实际上这些假设仅在一些特殊的实例中成立。一些自动化的证明器限制了添加的假设的使用，它们或是强制用户使用那些已经通过工具证明的定理去证明每条添加的假设，或是把使用额外假设的证明标记为“不安全”。

我们下面用验证一个算法的经验来总结有关演绎验证优点和缺点的讨论。SPIN 中使用的偏序约简算法 [67] 是用定理证明器 HOL 来形式化验证的 [26]。由于该算法很复杂，并且经常用在模型检验工具中来验证或是发现错误，因此人们认为其重要程度已经值得形式化验证所需的时间和努力。该过程花费了一个人十周的工作时间，产生了超过 7000 行的 HOL 证明。进行这项证明工作是值得的，因为该算法得到了扩展，可用于验证同时包含硬件和软件的嵌入式系统。然而，一年之后，人们发现当算法的特定的实现和 SPIN 中的“即时” DFS 一起工作时会引发一个错误 [68]。这个特别的经验证明了如下已经在本书中讨论过的观点：

- 演绎验证是非常耗时且相当复杂的。
- 可能需要对算法进行抽象以便于验证。
- 抽象版本和实际实现间的差异意味着即便证明是有效的，代码仍有可能包含错误。
- 模型检验工具和其他软件一样可能包含错误，其本身也需要验证和测试。

7.8 证明系统的正确性和完备性

给出一个证明系统，我们想要保证只有正确的断言能被证明。这被称为证明系统的*正确性*（soundness）。当一个证明系统由一组公理和证明规则给定时，其正确性通过以下两点来证明：

- 所有公理产生的断言都是正确的。
- 对每一个证明规则，如果前提成立，那么结论也成立。

例如，研究一下 Hoare 证明系统，也许有人能使自己确信其公理和证明规则在上述意义下是正确的。然而这该如何形式化验证呢？一种可能性是 Hoare 公理和证明规则*按定义*是正确的，即它们定义了诸如顺序、while 循环和 if-then-else 选择等编程结构如何执行。

一种不同的方法是用一些数学对象来形式化地定义编程语言的语义，例如将每个程序映射到一组状态序列上，每个序列代表一个执行。我们需要证明这种定义和证明系统之间的一致性。这里同样存在程序及其模型间的差异，因为实际程序的行为可能会不同。

对证明系统在多大程度上能够证明一个正确的断言做一个现实的评估也是非常重要的。证明系统的最终目标是成为*完备的*系统，换句话说就是要能够证明每一个用规约机制表示的正确的断言。有一些技术难题阻碍了程序验证证明系统达到这一目标，特别是对于基于一阶逻辑的系统：

- 在某些域上的基础性逻辑可能本身就不具备一个完整的公理和证明规则集合（参见 3.6 节）。我们经常想要使用自然数域上的一阶逻辑。Gödel 著名的不完备定理表明了自然数的完备的证明系统是不存在的。因此，在 Hoare 逻辑中，在试图证明所需的一阶蕴含式时我们可能会遇到困难，例如在左强化或右弱化规则中。
- 证明所需的关联于程序某些位置上的不变式或是中间断言可能不能使用基础的（一阶）逻辑来表达。例如，假设程序使用了某种关系 R 和它的*传递闭包*（transitive closure）

R^*，即 xR^*y 仅当有一组从 x 到 y 的元素序列，其中相连的元素由 R 关联。可以证明一阶逻辑不能表达 x 和 y 是由一个给定关系的传递闭包关联的 [38]（尽管在一阶逻辑中要表达一个关系 R 在传递关系下是闭合的是相当简单的）。

在大多数情况下，这些技术难题导致了完备的程序正确性的证明系统是不存在的。然而，研究人员发明了一个更弱化的概念，叫做证明系统的*相对完备性*（relative completeness）。该概念形式化了这样一个属性：任何导致证明失败的原因仅限于上面的两条原因，和处理程序的证明系统无关。当在下面两种情况下任何正确的断言都能被证明时，一个证明系统就是相对完备的：

1. 每个证明中需要的正确的（一阶）逻辑断言都已经作为一条公理包括在证明系统中。换句话说，存在一个无所不知的*判定规则*（oracle）（例如可以是人）来负责决定一个断言正确与否。

2. 每个我们需要的不变式或中间断言都是可表达的。

有这些（有时候是不现实的）假设就可以证明如 Hoare、Floyd 和 Manna 以及 Pnueli 证明系统的相对完备性。第一眼看去这可能是个无用的数学练习，或是证明系统的开发人员企图将可能的失败的责任转嫁给其他研究人员。然而事实不是这样的，相对完备性是一个重要而深刻的概念，因为：

- 它能识别验证中出现的问题到底在哪里，而且有助于描述哪些是可能的，哪些是不可能的。
- 它能提出规避问题的方法。例如当证明不需要乘法时，可以使用 Presburger 算术。它仅包含加法、减法和常量乘法，是自然数上的可判定理论。不过注意乘法可用重复的加法来编写。
- 它给出了正确性证明存在的希望。尽管不可能证明所有的基础逻辑和域的正确定理，但一个写好的程序很可能是基于已证明的属性。否则，就没有好的理由去相信它确实是对的。

7.9　组合性

我们渴求能够分别验证系统的不同部分，然后将证明合并起来。这叫做*组合验证*（compositional verification）。对顺序程序，可以把 Hoare 证明系统看成是组合的：要证明顺序组合 $\{\varphi\}S_1; S_2\{\psi\}$，可以通过找到既是 S_1 的后置条件又是 S_2 的前置条件的断言 η，然后分别证明 $\{\varphi\}S_1\{\eta\}$ 和 $\{\eta\}S_2\{\psi\}$ 来实现。很自然我们想找到 Hoare 证明系统的一个扩展，使得能够以组合的方式来处理并发程序。理想情况下，我们想把分开的不同进程的证明组合成一个完整系统的证明。我们现在说明这一目标会带来一些困难。

假设我们要证明对某个并发系统 $P_1 \parallel P_2$，属性 $\varphi \wedge \psi$ 成立，其中 φ 只涉及 P_1 中的变量，ψ 只涉及 P_2 中的变量。一种直接的尝试是证明当 P_1 和 P_2 分开执行时，φ 对 P_1 成立，ψ 对 P_2 成立。这是很难成功的：在大多数程序中，P_1 和 P_2 之间的交互会对它们的行为造成重要的影响。单个孤立进程可能甚至都不能运行完（例如，它可能会永久等待其他进程的消息，或是其他进程会改变某个变量的值），并且当和其他进程一起运行时它的行为会不同。

另一种组合验证的尝试是用一个更小、更简单的进程 P_ψ 来代替 P_2，P_ψ 已被证明满足 ψ，并且证明 $P_1 \parallel P_\psi$ 满足 φ。然后做对称的证明，用 P_φ 来代替 P_1，证明 ψ 对组合成立。然而，这种证明方案的论证是循环的，可能会导向错误的结果。

例如，假设 φ 声明了变量 x 的值最终从 0 改变为 1，ψ 声明了变量 y 的值最终从 0 改变为 1。现在，假设 P_1 的结构是它首先等待一个来自 P_2 的消息，该消息意味着 y 的值在 x 的值改变前被改变了。类似地，P_2 在等待一个消息，该消息意味着 x 的值在 P_2 改变 y 前被改变了。显然，$P_1 \parallel P_2$ 不满足 $\varphi \wedge \psi$。根据上面的证明策略，我们可以用一个进程 P_ψ 来代替 P_2，P_ψ 不用等待就

能把 y 从0改为1。那么 $P_1 \| P_\psi$ 满足 φ。类似地，我们可以用一个进程 P_φ 来代替 P_1，P_φ 不用等待就能把 x 从0改为1。那么 $P_\varphi \| P_2$ 满足 ψ。这能导致关于 $P_1 \| P_2$ 错误的结论。

7.10 演绎验证工具

Manna-Pnueli 证明系统的实验可通过 STeP（Stanford Theorem Prover）工具来进行。此工具可通过 http://rodin. stanford. edu 获得。

TLV 系统可被用于演绎式程序验证（也可用于模型检验）。它可以通过 http://www. wisdom. weizmann. ac. il/～verify/tlv 获得。

演绎式定理证明，如 3.11 节中提到的那些，也可以被用来验证程序的属性。

注意：使用形式化方法系统经常需要填写和发送一张授权表格并遵守一些使用条款。

7.11 扩展阅读

以下两篇综述 Hoare 验证方法的简练文章由 K. R. Apt 撰写：

K.R. Apt, Ten years of Hoare's logic: A survey, part I, Transactions on Programming Languages and Systems, 3(4), 1981, 431–383.

K.R. Apt, Ten years of Hoare's logic: A survey, part II: Nondeterminism, Theoretical Computer Science 28, 1984, 83–109.

关于 Hoare 风格的演绎式定理证明有几本书籍，包括：

N. Francez, *Program Verification*, Addison Wesley, 1992.

K. R. Apt, E.-R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991 (second edition, 1997).

F.B. Schneider, *On Concurrent Programming*, Springer-Verlag, 1997.

后者同时也包含了 Manna-Pnueli 风格的验证方法。一本全面的关于安全属性的时序验证的书籍是：

Z. Manna, A. Pnueli, *Temporal Verification of Reactive Systems: Safety*, Springer-Verlag, 1995.

Kröger 的书也介绍了时序验证方法。

F. Kröger, *Temporal Logic of Programs*, EATCS Monographs on Theoretical Computer Science, Springer-Verlag, 1992.

进程代数与等价关系

“……让我想想看：今天早晨我起床的时候是不是还是原样？我有点觉得，我能记得有点儿不同的感觉。不过我要是变得不一样了，那么下一个问题是：我到底是谁呢？……”

刘易斯·卡洛尔《爱丽丝漫游奇境记》

进程代数是对系统行为建模的形式化表述方法。进程代数技术和算法可以用来表示两个系统（模型）之间存在某种精确定义的关系，例如一个系统（模型）可以模拟另一个系统（模型）。这可以用来证明某个系统是另外一个系统的实现或精化。基于进程代数的手工或自动证明两个系统间存在某种特定关系的技术可用于逐步精化式的软件开发过程。在这样一个过程中，开发从产品规约开始，经过逐步的精化，最终得到实际的代码；精化的各个阶段，通过确保等价关系来保证精化过程的正确性。精化同时也允许（开发人员）使用一种层次化的结构来描述一个系统，在该层次结构中，每一层同上一层之间都有严格的形式化等价关系。

由于进程代数经常用于不同系统间的比较，因此如何为系统间选取适当的对应准则就变得非常重要。这个选择会直接影响在系统上可以做的观测和实验，以及在怎样的抽象层次上对系统进行描述。

下面的例子说明了为什么在多个系统间进行比较时，可能涉及多个不同的准则。考虑如图 8.1所示的两个转换系统。左侧的系统首先执行动作 α，然后选择执行动作 β 或是执行动作 γ。无论选择执行哪个动作，接下来它都会执行 τ，并返回到系统的初始状态。而右侧的系统，在执行动作 α 前首先做一个非确定性的选择。根据选择结果的不同，在执行动作 α 后，要么只允许执行动作 β，要么只允许执行动作 γ。之后，在两种情况下，都允许执行动作 τ 并返回到系统的初始状态。

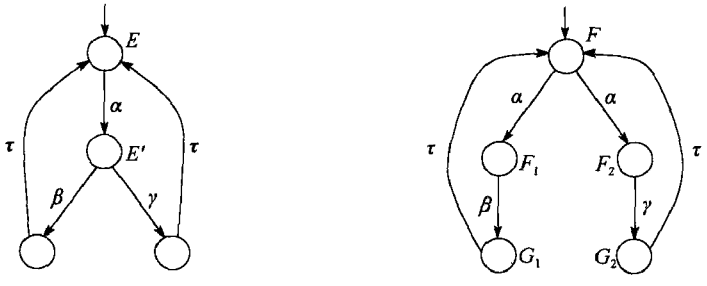


图 8.1　两个简单系统

如果上述两个转换系统分别代表两台自动贩卖机（自动贩卖机在进程代数中经常被当做例子）。其中假设 α 表示“付 25 美分”、β 表示“选择购买巧克力”、γ 表示“选择购买糖果”。在这种情况下，用户会更倾向于选择左侧系统所描述的机器。因为当付过 25 美分后，用户可以在“购买巧克力”和“购买糖果”间做出选择。而对于右侧系统所描述的机器，机器抢先做一个内部的非确定性选择，接下来用户只能根据这个选择的结果来购买“巧克力”或“糖果”当中的一种。但如果图 8.1 的两个系统表示的是自动邮票售卖机，其中 β 表示售出一枚蓝色的 33 美分邮票，而 γ 表示售出一枚红色的 33 美分邮票。（不考虑机器卖光所有的巧克力、糖果或者邮票的情

况。）这时，用户并不关心选择蓝色的邮票还是红色的邮票（除非该用户是集邮爱好者），因此可以认为上述两个机器是彼此等价的。

进程代数最初用于并发系统的理论研究，仅仅采用了最小化的符号表示。这使它在对实际的系统建模时略显吃力，需要加以扩展。本章的8.9节详细介绍了对进程代数的一种扩展方法。进程代数的方法对系统建模提供了新的途径。例如，可以利用进程代数处理非确定性和并发性之间的关系。

使用进程代数方法描述和分析系统行为的思想是由不同的学者分别提出的，其中以 Hoare [64]、Milner [100] 和 Park [112] 最为著名。在进程代数理论以及一系列基于进程代数理论的系统比较和系统验证工具得到发展的同时，模型检验技术也得到了相同程度的发展。事实上，在形式化方法领域中有一个有趣的现象，就是模型检验技术在北美学术界受到更多的关注，而进程代数在欧洲得到了广泛的支持。

8.1 进程代数

在进程代数中，agent（进程实体）是对一个系统的抽象描述。进程代数通常给出对 agent 的语法描述。agent 在这里被描述成项（类似于一阶谓词逻辑中的项），使用了顺序运算符、并发组合运算符以及非确定性选择运算符等对 agent 的行为加以刻画。进程代数的语义定义描述了在运算过程中 agent 的演化（例如前进或是变化）过程。各个运算符的性质以及它们之间的相互作用也是进程代数的主要研究内容。例如，研究非确定性选择运算符是否满足结合律与交换律（参见4.9节）。

进程代数的另一个重要组成部分是一个比较准则集，即 agent 之间的等价关系。当两个agent 的行为非常相似以至于我们不需要对它们加以区分时，则可判定这两个 agent 间存在等价关系。比较（结果）可以作为一种正确性准则，用于验证某系统的设计与规约间是否满足给定的关系。已有工具，如 Concurrency Workbench [31]，可以检验在给定的比较准则下两个 agent 之间存在某种关系。

在进程代数中通常没有显式的系统状态的概念，以映射变量到数值。一个 agent 本身就可以代表系统的一个状态，它刻画了从当前点开始，经过执行某个动作，最终演化为另一个 agent。因此，进程代数更关注系统的动作，而不是系统的状态。设 Act 为所有可见动作组成的集合，对于 Act 中的每个动作 α，都存在着一个与之对应的互补动作（co-action）$\bar{\alpha}$，满足 $\bar{\bar{\alpha}}=\alpha$。举例来讲，$\alpha$ 和 $\bar{\alpha}$ 可能代表一次同步通信过程，其中 α 表示接收（输入）动作，$\bar{\alpha}$ 表示发送（输出）动作。

除了在集合 Act 中的可见动作以外，还有一个特殊的不可见（内部或者静默）动作 τ，τ 的互补动作即为 τ 本身，即 $\bar{\tau}=\tau$。这样的动作 τ 可以表示一些内部的无需为外部感知的动作。在图8.1的例子中，两个系统的可见动作集 Act 均为 $\{\alpha, \beta, \gamma\}$。在进程代数中，某些等价关系需要区分可见动作和不可见动作。相应地，当采用实验的方法比较不同的 agent 时，很难控制系统中的不可见动作。

一个事件，记作 $E_1 \xrightarrow{\alpha} E_2$，表示 agent E_1 通过执行动作 α 演变成了另一个 agent E_2，其中 $\alpha \in Act \cup \{\tau\}$。如果存在 E_2 使得 $E \xrightarrow{\alpha} E_2$，则称动作 α 是在 agent E_1 上允许的（enabled）。如果 $E_1 \xrightarrow{\alpha_1} E_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} E_n$，则称 E_1 通过执行动作序列 $\sigma=\alpha_1\alpha_2\cdots\alpha_{n-1}$ 可以导出 E_n，也可以写作 $E_1 \xrightarrow{\sigma} E_n$。

在图8.1所示的例子中，左侧的转换系统表示 agent E 可以通过执行动作 α 演化为另一个 agent E'，该过程所对应的事件为 $E \xrightarrow{\alpha} E'$。而在右侧的转换系统中，agent F 通过执行动作 α 演

化为 F_1 或 F_2 中的一个，该过程所对应的事件为 $F \xrightarrow{\alpha} F_1$ 和 $F \xrightarrow{\alpha} F_2$，其中左边的 F_1 只能执行动作 β，而右边的 F_2 只能执行动作 γ。

与转换系统的模型类似，可以将 agent E 的执行定义为一个最大序列（如可能是一个无限序列或是不可扩展的有限序列）$E=E_1 \xrightarrow{\alpha_1} E_2 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_{n-1}} E_n$，使得对任意的 $i \geqslant 1$，$E_i \xrightarrow{\alpha_i} E_{i+1}$ 是一个事件。一个执行是有限的，仅在其对应的序列最后的 agent 上必然不存在可执行的动作。同样，可以定义一个图来描述 agent E，图中的节点表示 E 所能演化出的 agent，图中的边表示事件，用动作来标记。在这张图中出现的 agent 也称作 E 的**配置**（configuration）。

一个**扩展事件**描述一个 agent 如何通过动作 τ 的一个（可能为空）序列，紧跟着一个或零个可见动作以及动作 τ 的一个（可能为空）序列，来进行演化。对一些 agent G_1，…，G_n，当存在不可见动作的一个（可能无关紧要的）序列

$$G \xrightarrow{\tau} G_1 \xrightarrow{\tau} G_2 \xrightarrow{\tau} \cdots \xrightarrow{\tau} G_n \xrightarrow{\tau} G'$$

时，我们表示为 $G \overset{\varepsilon}{\Longrightarrow} G'$。那么，扩展事件 $E \overset{\alpha}{\Longrightarrow} E'$ 表示存在某 agent F 和 F' 使得

$$E \overset{\varepsilon}{\Longrightarrow} F \xrightarrow{\alpha} F' \overset{\varepsilon}{\Longrightarrow} E'$$

例如，如在图 8.1 右边，有 $F_1 \overset{\beta}{\Longrightarrow} F$ 和 $G_2 \overset{\alpha}{\Longrightarrow} F_1$。$E \xrightarrow{\tau^\infty}$ 表示从 agent E 可以执行无限多个内部动作这样一个事实。这里，我们称 E 是**发散的**（diverge）。理解扩展事件的一种方法是把它看成是一个实验（experiment）中的最小可见部分。考虑一个不能看到动作 τ 的系统的观察者，则 $F_1 \overset{\beta}{\Longrightarrow} F$ 或 $G_2 \overset{\alpha}{\Longrightarrow} F_1$ 表示关于这个系统所能得到的外部观察。

8.2　通信系统的演算

本章使用由 Milner [100] 定义的 CCS（Calculus of Communicating Systems）的语义和语法（除了用并发组合“$\|$”代替了“$|$”之外）。令 α 为 $Act \cup \{\tau\}$ 中任意的一个元素，R 为 Act 的一个子集，f 为一个动作到动作的映射，C 表示一个 agent 变量。agent 可以用 BNF 定义如下：

$$agent ::= \alpha.agent \mid agent + agent \mid agent \| agent \mid agent \setminus R \mid agent[f] \mid agent \mid C \mid 0$$

在 CCS 的各个运算符中，限制符（“$\setminus$”）拥有最高的优先级，其次是映射（“$[]$”），接下来是顺序前缀（“$.$”），然后是并发组合（“$\|$”），最后是非确定性选择（“$+$”）。需要注意的是前缀运算符“$.$”并不表示顺序连接（如编程语言中的“;”）：通常对于两个进程代数表达式 E_1 和 E_2，不能写成 $E_1.E_2$，但是 $\alpha.E_2$ 这种写法是正确的，其中 α 是一个动作。我们用 agent “0” 表示终止，它不允许任何动作（包括 τ）。通常在描述一个 agent 的时候我们不会使用如“$.0$”或者“$.(0 \| \cdots \| 0)$”的表述方式。

agent 的语义通过结构化操作语义（Structural Operational Semantics，SOS）规则来描述，很像是证明规则。这乍看上去有些奇怪。比较在一阶谓词逻辑中，公理和推理证明是由语法对象构成的，而逻辑语义的证明是通过产生正确的公式进行的。类似地，Hoare 证明系统（见 7.4 节）也可以被看做是程序的公理语义表示。在证明代数中，用证明规则来描述一个 agent 的演化过程，我们可以通过它来验证一个事件的性质，即可以通过推理证明得到一个 agent 演化为另一个 agent 所执行的动作。

8.2.1　动作前缀

令 $\alpha \in Act \cup \{\tau\}$，$E$ 为一个 agent，则 agent $\alpha.E$ 表示：首先执行一个动作 α，接下来的动作由 agent E 描述。亦可用如下公理表示：

$$\alpha.E \xrightarrow{\alpha} E \tag{8.1}$$

可以用上述公理去验证如下事件：

$$\alpha.(\beta.(\delta \parallel \bar{\delta}) + \gamma) \xrightarrow{\alpha} \beta.(\delta \parallel \bar{\delta}) + \gamma$$

8.2.2 选择

执行 $E+F$ 通过在 E 和 F 之间做非确定性的选择，然后根据所选的 agent 执行接下来的操作来完成。可用如下两条 SOS 规则表示，它们分别对应选择根据 E 或根据 F 执行的情况。

$$\frac{E \xrightarrow{\alpha} E'}{E+F \xrightarrow{\alpha} E'} \qquad \frac{F \xrightarrow{\alpha} F'}{E+F \xrightarrow{\alpha} F'}$$

我们可以用动作前缀公理去验证如下事件：

$$\beta.(\delta \parallel \bar{\delta}) \xrightarrow{\beta} \delta \parallel \bar{\delta} \tag{8.2}$$

继而，我们可以用事件（8.2）作为前提，利用左侧选项的 SOS 规则来验证如下事件：

$$\beta.(\delta \parallel \bar{\delta}) + \gamma \xrightarrow{\beta} \delta \parallel \bar{\delta}$$

注意在这种情况下，从 $\beta.(\delta \parallel \bar{\delta})+\gamma$ 执行 β 意味着选择运算符左侧的部分被选中，执行 γ 则意味着选择运算符右侧的部分被选中。但是对于 agent $\beta.\delta+\beta.\gamma$ 来说，执行 β 则可能对应着选中左侧或右侧部分的两种情况。

8.2.3 并发组合

agent $E \parallel F$ 表示 agent E 与 agent F 的并发组合。可以按如下三种情况执行：

- agent E 执行某个动作 $\alpha \in Act \cup \{\tau\}$，agent F 保持不变：

$$\frac{E \xrightarrow{\alpha} E'}{E \parallel F \xrightarrow{\alpha} E' \parallel F}$$

- agent F 执行某个动作 $\alpha \in Act \cup \{\tau\}$，agent E 保持不变：

$$\frac{F \xrightarrow{\alpha} F'}{E \parallel F \xrightarrow{\alpha} E \parallel F'}$$

- agent E 执行某个动作 $\alpha \in Act$，而 agent F 执行它的互补动作 $\bar{\alpha}$。结果：E 与 F 在这个动作上实现同步，而这个同步执行用不可见动作 τ 标记：

$$\frac{E \xrightarrow{\alpha} E', F \xrightarrow{\bar{\alpha}} F'}{E \parallel F \xrightarrow{\tau} E' \parallel F'}$$

我们可以用动作前缀公理去验证如下事件：

$$\delta \xrightarrow{\delta} 0 \tag{8.3}$$

和

$$\bar{\delta} \xrightarrow{\bar{\delta}} 0 \tag{8.4}$$

因此，把事件（8.3）、（8.4）作为后一个并发组合规则的前提，我们可以得到事件：

$$\delta \parallel \bar{\delta} \xrightarrow{\tau} 0 \parallel 0 \tag{8.5}$$

8.2.4　限制符

用某个动作集 $R \subseteq Act$ 来限制 agent E，不允许从 E 执行一个动作序列得到的任意 agent 执行 R 中的任意动作或其相应的互补动作。

$$\frac{E \xrightarrow{\alpha} E', \alpha, \bar{\alpha} \notin R}{E \setminus R \xrightarrow{\alpha} E' \setminus R}$$

我们用 agent 变量 C 表示 agent $\delta \| \bar{\delta}$，现在我们可用第一个并发组合规则中的事件（8.3）得到事件：

$$C \xrightarrow{\delta} 0 \| \bar{\delta}$$

上述公式表示并发 agent C 可以与一个外部的互补动作 $\bar{\delta}$ 交互，作为内部动作 δ 和 $\bar{\delta}$ 同步执行的替换。同理，我们可以证明

$$C \xrightarrow{\bar{\delta}} \delta \| 0$$

因此，agent C 可以与外部动作 δ 或 $\bar{\delta}$ 交互。

我们希望避免上述情况的发生，只希望 δ 和 $\bar{\delta}$ 作为内部动作在 agent C 的范围内进行交互，而对外不可见。我们可以通过使用并发 agent $C \setminus \{\delta\}$ 来实现。这是一个对外隐藏动作 δ（及其互补动作 $\bar{\delta}$）的 agent，对于这样的 agent 不能出现标记为 δ 或 $\bar{\delta}$ 动作的事件。

我们发现利用第三条并发组合规则，可以得到 $C \xrightarrow{\tau} 0 \| 0$（参见式（8.5））。在单事件集合 $\{\delta\}$ 上的限制使得 δ 和 $\bar{\delta}$ 之间的交互只能在 agent C 的内部进行，因为这个限制只是针对并发 agent C 而不针对 C 内部的并发组件。当交互发生时，这个 δ 和 $\bar{\delta}$ 的联合动作在外部被标记为不可见动作 τ，内部动作 τ 不应该，也不会被隐藏起来。当我们把事件（8.5）作为限制规则中的前提时，可以得出以下结论

$$C \setminus \{\delta\} \xrightarrow{\tau} 0 \| 0 \setminus \{\delta\} \tag{8.6}$$

注意，在这里 τ 是 $C \setminus \{\delta\}$ 上唯一允许的动作。

8.2.5　重标记

令 $m : Act \to Act$ 为一个映射，满足 $m(\bar{\alpha}) = \overline{m(\alpha)}$，即如果 m 将 α 映射到 β，它也将 $\bar{\alpha}$ 映射到 $\bar{\beta}$（因此，如果 $m(\alpha)=\bar{\beta}$，那么 $m(\bar{\alpha})=\beta$）。所以，用 $E[m]$ 表示重标记动作。

$$\frac{E \xrightarrow{\alpha} E'}{E[m] \xrightarrow{m(\alpha)} E'[m]}$$

举一个简单的例子，考虑映射 $m = \{\alpha \mapsto \beta, \beta \mapsto \alpha\}$。用事件 $\alpha \xrightarrow{\alpha} 0$ 作为重标记规则的前提，我们可以得到事件 $\alpha[m] \xrightarrow{\beta} 0[m]$ 作为结论。通常为了方便，将重标记写作 $E[x_1/y_1, x_2/y_2, \cdots, x_n/y_n]$，表示将 x_1 映射为 y_1，将 x_2 映射为 y_2，以此类推。在方括号中没有出现的动作映射到这个动作本身。

8.2.6　等式定义

目前所定义的 CCS 还不足以对有无限行为的 agent 进行描述。为创建这样的 agent，CCS 也支持在等式集合上的递归定义。一个 agent 变量因此可用包含其名字的术语来定义。

等式定义用如下公式表达：$A \triangleq E$，其中 A 为 agent 变量，而 E 是 agent。表达式 E 或许会包

括 agent 变量 A。多个 agent 变量可以通过互相引用定义，例如，$A \triangleq \alpha.B$ 和 $B \triangleq \beta.A$。这时，agent A 可通过执行动作 α 变成 agent B；agent B 可通过执行动作 β 变成 agent A，以此类推。

对于图 8.1 中的两个系统，左侧的系统可以用如下的公式定义：

$$E \triangleq \alpha.E'$$

$$E' \triangleq \beta.\tau.E + \gamma.\tau.E$$

等价地，可用一行表示为

$$E \triangleq \alpha.(\beta.\tau.E + \gamma.\tau.E)$$

注意，这里中间 agent 变量 E' 在后面的表示中被消去了。图 8.1 右侧的系统可以定义为：

$$F \triangleq \alpha.F_1 + \alpha.F_2$$

$$F_1 \triangleq \beta.G_1$$

$$F_2 \triangleq \beta.G_2$$

$$G_1 \triangleq \tau.F$$

$$G_2 \triangleq \tau.F$$

等价地，可用一行表示为

$$F \triangleq \alpha.\beta.\tau.F + \alpha.\gamma.\tau.F$$

要谨慎地使用递归定义。例如，考虑 $A \triangleq B$ 和 $B \triangleq A$ 的情况。如果我们把这些等式用于定义，我们只知道 A 与 B 是相同的，但是对于它们的行为没有任何限制。为了避免这种情况的发生，对于等式定义形式通常加以一些限制，如定义的右边必须以一个前缀动作或者是一个非确定性选择为开头，其中每个组件都带有一个前缀动作。

令 E 为某个 agent，定义 $A \triangleq E$ 的行为的规则如下：

$$\frac{E \xrightarrow{\alpha} E', A \triangleq E}{A \xrightarrow{\alpha} E'}$$

将等式和并发组合结合起来，我们可以定义出包含无限个并发组件的 agent。例如，令 $A \triangleq \alpha \| \beta.A$ 表示 agent A 可以通过执行 n 次 β 动作而分裂成 $n+1$ 个并发组件，其中 n 个并发组件仅执行动作 α 并终止，而第 $n+1$ 个并发组件继续分裂成两个并发组件。此次执行的前缀如图 8.2 所示。

下面以事件 $\alpha \| A \xrightarrow{\beta} \alpha \| (\alpha \| A)$ 的推导为例，来说明这个序列是如何形成的。

(1) $\beta.A \xrightarrow{\beta} A$ [动作前缀]

(2) $\alpha \| \beta.A \xrightarrow{\beta} \alpha \| A$ [并发组合和 (1)]

(3) $A \xrightarrow{\beta} \alpha \| A$ [等式定义和 (2)]

(4) $\alpha \| A \xrightarrow{\beta} \alpha \| (\alpha \| A)$ [并发组合和 (3)]

如果一个 agent 能演化为有限多个不同的 agent，那么它表示一个有限状态系统，否则它表示的就是一个无限状态系统。显然，agent $A \triangleq \alpha \| \beta.A$ 所表示的是一个无限状态系统。

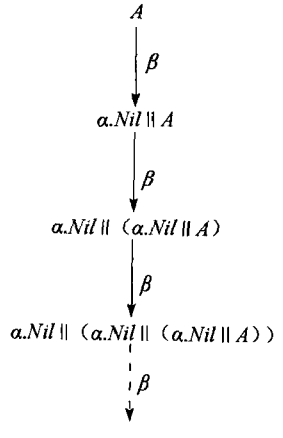


图 8.2 重复执行 $A \triangleq \alpha \| \beta.A$ 中的 β

8.2.7　agent 0

没有与 agent 0 相对应的公理或者规则，因此，这样的 agent 上不允许任何动作，也无法演化成其他的 agent。

图 8.3 为 agent $\alpha.(\beta.(\delta \parallel \bar{\delta})+\gamma)$ 的示意图。利用前面提到的公理和证明规则，某些事件已经得到验证。因为没有使用等式定义，所以在该 agent 上不会有无限执行，相应地，图中也不会出现环。

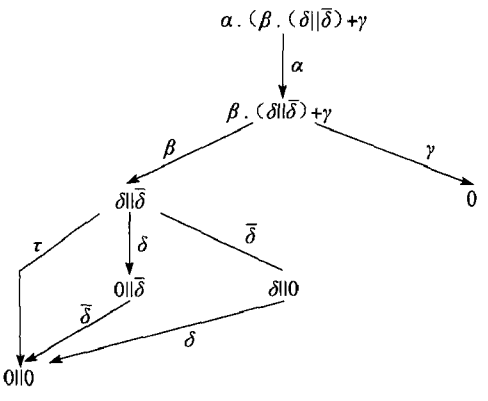


图 8.3　$\alpha.(\beta.(\delta \parallel \bar{\delta})+\gamma)$ 的示意图

8.2.8　传值 agent

为了描述更实际的系统，动作和 agent 变量可以用参数表示。例如，对于 agent $a(x).P$ 首先执行输入一个值赋给 x 的操作，而 agent $\bar{a}(y).Q$ 首先要执行将 y 的值输出的操作。通过这样的一次交互在 agent P 中实现了 x 与 y 的值绑定。

例如，对于如下 agent：

$$buy(x).(\overline{insure}(x).(\overline{drive}(x)))$$

在这个例子中，输入某个值给 x，接着先后在动作 $\overline{insure}(x)$ 和 $\overline{drive}(x)$ 中将这个值输出。更多关于传值进程代数的信息请参见 [21]。

8.3　示例：Dekker 算法

下面我们将用 CCS agent 来表示在 4.6 节中提到的 Dekker 互斥算法。

首先是关于状态在进程代数中的表示问题。在 CCS 中，一个状态就是一个 agent，即一个可演化成其他表达式的表达式。为了这样表示变量的值，首先要为每个具有两个可能值（如 $c1$，$c2$ 和 $turn$）的变量构造一个具有两个状态的自动机。这些状态分别对应变量的值。图 8.4 为变量 $c1$ 所对应的自动机，该自动机使用了如下的互补动作。

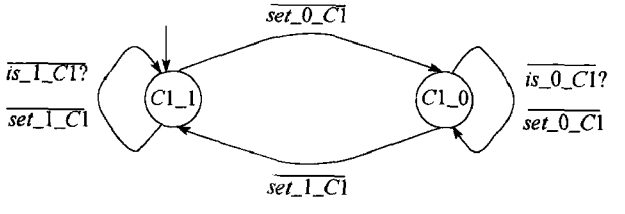


图 8.4　表示二值变量 $c1$ 的自动机

$\overline{set\_0\_C1}$：将 $c1$ 的值设为 0。自动机如果不在状态 C1_0（指 $c1$ 为 0）上，则跳转到该状态上。

$\overline{set\_1\_C1}$：将 $c1$ 的值设为 1。自动机如果不在状态 C1_1（指 $c1$ 为 1）上，则跳转到该状态上。

$\overline{is\_0\_C1}$：这个动作是状态 C1_0 上的自循环，它的作用是检查 $c1$ 的值是否为 0。

$\overline{is\_1\_C1}$：这个动作是状态 C1_1 上的自循环，它的作用是检查 $c1$ 的值是否为 1。

$c1$ 所对应的自动机被实现为从 C1_1 开始的 agent。这个 agent 可以与使用上述互补动作所对应的各种动作的 agent 交互。agent C1_1 可以通过执行动作 $\overline{set\_0\_C1}$ 演化为 agent C1_0，agent C1_0 可以通过执行动作 $\overline{set\_1\_C1}$ 演化为 agent C1_1。除此之外，另外还有代表进程 $P1$ 或者 $P2$ 的agent可以将 $c1$ 设置为 0 或 1，或者在 $c1$ 的某个值上等待，直到其改变。

我们定义如下三个进程实体（agent），分别表示 $c1$，$c2$，$turn$ 三个变量。

*c*1

$$C1\_0 \triangleq \overline{is\_0\_C1?}.\ C1\_0 + \overline{set\_1\_C1}.\ C1\_1 + \overline{set\_0\_C1}.\ C1\_0$$

$$C1\_1 \triangleq \overline{is\_1\_C1?}.\ C1\_1 + \overline{set\_0\_C1}.\ C1\_0 + \overline{set\_1\_C1}.\ C1\_1$$

*c*2

$$C2\_0 \triangleq \overline{is\_0\_C2?}.\ C2\_0 + \overline{set\_1\_C2}.\ C2\_1 + \overline{set\_0\_C2}.\ C2\_0$$

$$C2\_1 \triangleq \overline{is\_1\_C2?}.\ C2\_1 + \overline{set\_0\_C2}.\ C2\_0 + \overline{set\_1\_C2}.\ C2\_1$$

turn

$$TURN\_1 \triangleq \overline{is\_1\_Turn?}.\ TURN\_1 + \overline{set\_2\_Turn}.\ TURN\_2 + \overline{set\_1\_Turn}.\ TURN\_1$$

$$TURN\_2 \triangleq \overline{is\_2\_Turn?}.\ TURN\_2 + \overline{set\_1\_Turn}.\ TURN\_1 + \overline{set\_2\_Turn}.\ TURN\_2$$

接下来定义进程 *P*1 和 *P*2。给这些进程编码有点棘手，因为 CSS 中没有顺序组合的运算符。这里根据变量值所做出的每个选择用选择运算符表示。每个选择都有一个前缀动作。例如，当我们查看 *c*1 的值是 0 还是 1 的时候，我们在动作 *is*_0_*C*1? 和动作 *is*_1_*C*1? 中作出选择；动作 *critical*1 和动作 *critical*2 表示两个进程在临界区内执行的动作；类似地，动作 *noncritical*1 和动作 *noncritical*2 表示两个进程在临界区外部执行的非互斥动作。

*P*1

$$P1 \triangleq noncritical1.\ set\_0\_C1.\ INSIDELOOP1$$

$$INSIDELOOP1 \triangleq is\_1\_C2?\ .\ REST1 + is\_0\_C2?\ .\ (is\_2\_Turn?\ .\ set\_1\_C1.\ INNERMOST1 + is\_1\_Turn?\ .\ INSIDELOOP1)$$

$$INNERMOST1 \triangleq is\_2\_Turn?\ .\ INNERMOST1 + is\_1\_Turn?\ .\ set\_0\_C1.\ INSIDELOOP1$$

$$REST1 \triangleq critical1.\ set\_1\_C1.\ set\_2\_Turn.\ P1$$

*P*2

$$P2 \triangleq noncritical2.\ set\_0\_C2.\ INSIDELOOP2$$

$$INSIDELOOP2 \triangleq is\_1\_C1?\ .\ REST2 + is\_0\_C1?\ .\ (is\_1\_Turn?\ .\ set\_1\_C2.\ INNERMOST2 + is\_2\_Turn?\ .\ INSIDELOOP2)$$

$$INNERMOST2 \triangleq is\_1\_Turn?\ .\ INNERMOST2 + is\_2\_Turn?\ .\ set\_0\_C2.\ INSIDELOOP2$$

$$REST2 \triangleq critical2.\ set\_1\_C2.\ set\_1\_Turn.\ P2$$

图 8.5 为并发 agent *P*1 的示意图，同样可得 agent *P*2（只需做一些如将 *C*1 换成 *C*2 的简单替换）。图 8.5 中标记的节点与上面所定义的 agent 变量相对应。

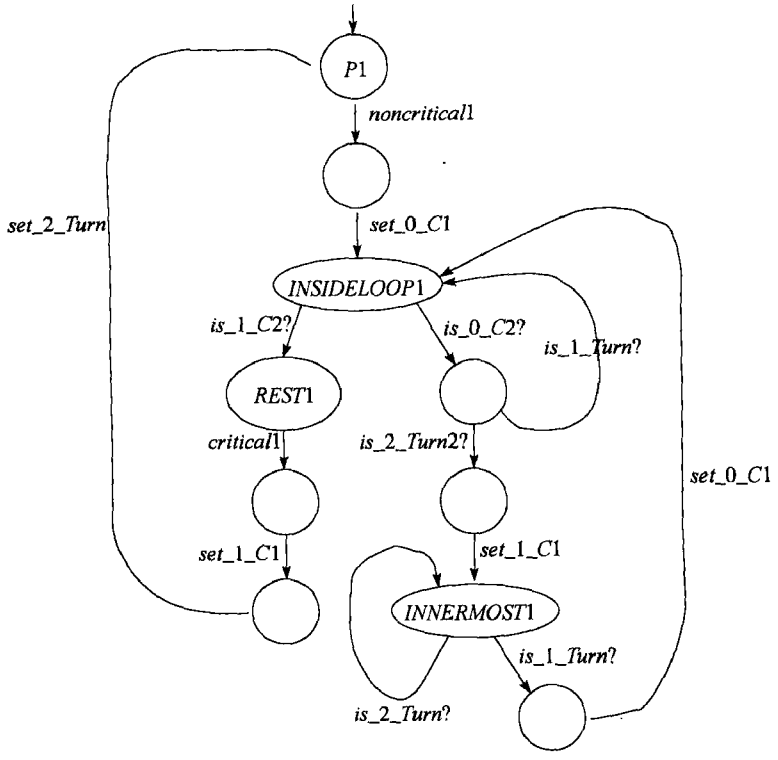


图 8.5　agent $P1$

Dekker 算法

在很多 CCS 所描述的系统中，某些动作外部观察者可能不感兴趣。因此对这样的动作要加以限制。这不仅仅是为了进程代数描述的简洁性，也是为了定义的安全性，即防止一些外部环境或者其他进程通过一些本来只应在内部执行的动作来与内部的进程进行交互。令 R 为如下动作集合：

$\{is\_0\_C1?, set\_0\_C1, is\_1\_C1?, set\_1\_C1, is\_0\_C2?, set\_0\_C2, is\_1\_C2?, set\_1\_C2, is\_1\_Turn?,$
$is\_2\_Turn?, set\_1\_Turn?, set\_2\_Turn\}$

则 Dekker 算法可以用进程代数表示为：

$$DEKKER \triangleq (C1\_1 \parallel C2\_1 \parallel TURN\_1 \parallel P1 \parallel P2) \setminus R$$

练习 8.3.1　用传值 agent 描述 Dekker 算法。

练习 8.3.2　用进程代数为 4.15 节中提到的通信协议建模。

练习 8.3.3　用进程代数为 4.6 节中提到的 Dijkstra 临时互斥算法建模。

8.4　建模问题

可以用 Dekker 算法的例子来说明系统建模中存在的一些问题。

考虑在 4.6 节所提到的并发进程的 wait 语句，如 wait until turn=1。这种建模方式对应忙等待（busy waiting）：进程 $P1$ 循环不断地在 $INNERMOST1$ 用动作 $is\_1\_Turn?$ 和 $is\_2\_Turn?$ 检查 $turn$ 的值。如果 $turn$ 值为 1，就继续将 $c1$ 的值通过执行动作 set_0_C1 设为 0；否则返回 IN-

*NERMOST*1。实际上，也可以采用另外一种等待方式：进程 *P*1 首先将自己挂起，当 *P*2 执行动作将 *turn* 的值变为 1 时，由操作系统唤醒 *P*1。运用这种建模方法，就可以去掉 *INNERMOST*1 上标记为 *is_2_Turn*? 的自循环，其结果可以用下面的表达式表示：

$$INNERMOST1 \triangleq is\_1\_Turn?.\ set\_0\_C1.\ INSIDELOOP1$$

这时，在 *INNERMOST*1 上不存在动作 *is_1_Turn*? 之外的选择。用忙等待的方式实现的 Dekker 算法可能会存在 *INNERMOST*1 上的永久等待，而在上述改进版本中则不会出现永久等待的情况，因为 *P*1 将自己挂起直至 *turn* 的值变为 1。

对非临界区建模也存在相似的问题。在将 4.6 节的算法翻译成进程代数的过程中，非临界区的操作被描述成一个单一的动作 *noncritical*1 或 *noncritical*2。然而，互斥算法在一个进程决定永远停留在临界区内的情况下也应该有效。为此，我们将表达式 *P*1（以及相应的 *P*2）修改为：

$$P1 \triangleq noncritical1.\ P1 + set\_0\_C1.\ INSIDELOOP1 \tag{8.7}$$

上述两种忙等待循环不应该延迟其他进程进入临界区。但是，这个属性的简单验证（如使用某些模型检验工具或者进程代数工具）却可能得出进程不能保证可以进入临界区的结论。造成这种结果的原因是验证工具通常对执行不做任何公平性假设。如果不做任何公平性假设，验证工具可能会返回如下反例：由于 *P*1 一直处于 *INNERMOST*1 循环中，因此 *P*2 也许无法进入临界区。此时，*turn* 的值是 2。在此情况下，*P*1 无限地执行 *is_1_Turn*? 动作。因为没有公平性假设，进程 *P*2 无法进入或者离开临界区，进而无法给 *turn* 赋值为 1，使 *P*1 走出 *INNERMOST*1 循环。

弱公平性假设可以保证在 *INNERMOST*1 循环中的进程 *P*1 等待 *turn* 变为 1 的时候，进程 *P*2 可以继续下去并将 *turn* 值赋为 1，以确保 *P*1 可以走出循环。同理，在弱公平性假设下，可以保证进程 *P*2 在满足条件的情况下可以进入临界区，即使 *P*1 一直在非临界区内（在式（8.7）的另一种定义下）。

但是很遗憾，有些验证工具缺乏强制保证整体公平性的能力，或者包含有规定的、不符合用户需求的公平性。这是因为公平性的假设会增加验证的复杂度 [78]。事实上，原始的 CCS 语义并没有做任何公平性假设。我们仍可以利用一个没有公平性保证的工具去验证某个时序属性 φ。无须预先建立公平性假设，我们可以在没有公平性假设的条件下验证属性 $\psi \rightarrowtail \varphi$。

练习 8.4.1 即使消除掉本节讨论的两个忙等待循环，仍需要公平性以便保证进程最终一定能进入临界区，当它希望这样做时。举一个出现上述问题的场景。一种找到这种场景的方式是使用模型检验或者进程代数工具（在第 6 章和本章最后都列举出了一系列上述工具）。提示：上述情况中包含了动作 *is_0_C2*? 和 *is_1_Turn*?。

8.5 agent 之间的等价性

agent 之间的比较很有用，例如，当其中一个 agent 表示一个系统或者一个实现，而另一个表示某种抽象的规约时。比较对于说明一个系统是另一个的精化也很有用。通过逐步构建更加细化的系统来完成开发时，这是一个重要的衡量标准。正如我们将看到的，有不止一种方法来比较 agent，选取正确的比较标准是非常重要的。不同比较标准之间的差异可能极其细微，下面的例子可以充分展示这一点。至于什么是合适的比较标准，目前还没有达成广泛的共识。

每个比较标准都可以用 agent 之间的等价关系给出，它是一种自反的、对称的、传递的关系（见 2.1 节）。因此，显示每对逐步细化的系统间的等价性保证了第一个（最粗略的，通常是规约）和最后一个（最细节化的，通常是实现）系统之间的等价性。

可以在不同的等价之间形成一种层次（一种偏序）。如果任意两个 agent 根据等价关系“$\equiv_1$”是等价的，同时根据等价关系“$\equiv_2$”也是等价的，那么等价“$\equiv_1$”比等价“$\equiv_2$”*更加精化*，反之则不需要。我们根据 van Glabbeek [53] 展示了他的进程代数等价关系间的部分层次。

8.5.1 迹等价

在进程代数中，迹（trace）是指可以从一个给定的 agent 执行的一个动作的有限序列。[⊖]令 $T(E)$为可从 agent E 执行的所有迹的集合。那么，当 $T(E)=T(F)$时，E 与 F 是迹等价的（trace equivalent），表示为 $E\equiv_t F$。

迹等价的定义有几个不同的变体。有一种可能是同时考虑有限和无限的迹。可以表明如果 E 和 F 是有限状态系统，那么 $T(E)=T(F)$也能说明 E 和 F 的无限迹的集合是一样的。证明在下一段中给出，读者可以安心跳过。

假定 $T(E)=T(F)$。考虑从 E 和 F 以及它们在树中的展开所得到的分支结构。在这些展开中，E 和 F 是各自的根，每个节点表示通过执行从相应的根到该节点的路径上的迹所得到的一个 agent。假定 E 有一个 F 所没有的无限迹 σ。因为 $T(E)=T(F)$，F 拥有 E 所拥有的每一个 σ 的有限前缀。我们将从 σ 的前缀构造一棵新的以 F 为根的树：这棵树包含 agent F，F 执行 σ 中的第一个动作 α 后所得到的那些 agent（注意，因为非确定性的原因，可能有多个从 F 执行 α 的方式），从这些 agent 再执行第二个动作所得到的那些 agent，依次类推。这是一棵无限树，因为对于无限的 σ 有无限多的前缀。每一个节点有有限多个直接后继，因为这是一个有限状态系统。根据 König 推论（见 2.3 节），必定存在某些始于新构建的树的根 F 的无限序列。根据该构造，这个序列必定被标记为 σ，从而形成矛盾。

如果进程代数 agent 被重新定义为允许无限多的直接后继（例如，通过允许一个无界的、参数化的非确定性选择运算符），那么迹等价就不意味着含有同样的无限迹的集合。由于无限多个后继的可能性，König 推论在这种情况下不成立。更进一步，甚至在允许有限多个后继的情况下，如果对执行加上额外的公平性约束，两个 agent 之间的迹等价也不意味着它们有同样的无限序列的集合。这些约束可以（比如）采用某种特定的配置强制每次执行无限多次地通过，类似于 Büchi 自动机中的接受条件。

另一种版本的迹等价要求在有同样的迹集合的基础上，agent 要有同样的*终止迹*的集合，即有限迹不能以一个被允许的动作扩展。这样带来了一种比迹等价要好的等价。

8.5.2 失败等价

agent E 的一个失败是指一个对 $\langle\sigma, X\rangle$，其中，σ 是这样的一个迹：从 E 开始执行 σ 得到某个 agent F，X 中的每一个动作对于 F 都不被允许。注意，失败的定义允许不在 X 中的动作对于 F 也是不被允许的。该定义的直接后果是，如果 $\langle\sigma, X\rangle$ 是 E 的一个失败，且 $Y\subseteq X$，那么 $\langle\sigma, Y\rangle$ 也是 E 的一个失败。需要注意的是即使 $\langle\sigma, X\rangle$ 是 F 的一个失败，那么根据非确定性，我们也可能根据 σ 从 E 开始执行相应动作并到达另外一个 agent $F'\neq F$，从此处可以执行 X 中的某些动作。令 $Fail(E)$为 agent E 的失败的集合。当 $Fail(E)=Fail(F)$时，我们说 E 和 F 是失败等

⊖ 这个定义可能会带来误解，因为此处的迹定义不同于并发理论中常用的由 Mazurkiewicz [96] 给出的迹定义。Mazurkiewicz 迹在字母表和该字母表上的二元关系之上定义，这个二元关系被称为*独立关系*。Mazurkiewicz 迹是一个包含这样序列的最大集合，这些序列是通过反复地计算单词中独立的字母而得到的。

价的，定义为 $E \equiv_{fl} F$。

容易看出失败等价是迹等价的一个精化：$Fail(E)$包含所有的对$\langle \sigma, \varnothing \rangle$，其中 σ 是 E 的一个迹。因此，$Fail(E)=Fail(F)$蕴含 $T(E)=T(F)$。接下来的例子将展示在迹等价下等价的 agent 在失败等价中有所区别。图 8.6 展示了两个含有同样的迹的 agent（甚至有相同的终止迹），但是它们的失败有所不同。左边的 agent $\alpha.(\beta+\gamma)$在执行 α 后总是可以执行 γ。右边的 agent $\alpha.\beta+\alpha.(\beta+\gamma)$有失败 $\langle \alpha, \{\gamma\} \rangle$，因为它可以选择左边的 α 分支，从该分支它不能执行 γ。

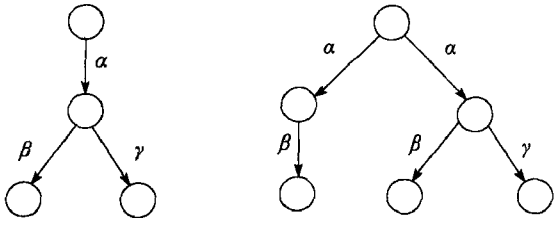


图 8.6 agent $\alpha.(\beta+\gamma)$和 $\alpha.\beta+\alpha.(\beta+\gamma)$

为了获取为什么这种区别可能很重要的直观感受，假定这两个 agent 描述的是简单的售货机，它们的动作如下：

α 投入一枚硬币。

β 选择一块巧克力。

γ 选择一颗糖。

机器 $\alpha.(\beta+\gamma)$允许使用者在投入一枚硬币后在一块巧克力和一颗糖之间做出选择。机器 $\alpha.\beta+\alpha.(\beta+\gamma)$在硬币投入后做的是非确定性选择。后面机器的一种非确定性选择是只允许使用者选择一块巧克力，而另一种选择允许使用者选择一块巧克力或者一颗糖。这个非确定性选择可以反映出一些隐藏的内部细节。例如，在投入硬币之后，后面的机器可能检查发现糖已经卖完了。

对这种等价的另一种定义方法出现在 [105] 中。它定义了对 $\langle \sigma, X \rangle$，称为必须对（must-pair）：从一个 agent E 开始执行动作序列 σ 之后所得到的每一个 agent，至少有一个在 X 中的动作是被允许的。从 E 出发的必须对的集合记为 $M(E)$。此处，如果 $\langle \sigma, X \rangle$ 是 E 的一个必须对，且 $Y \supseteq X$，那么 $\langle \sigma, Y \rangle$ 也是 E 的一个必须对。那么，仅当 $M(E)=M(F)$时，$E \equiv_{m} F$。

我们将说明失败等价$\equiv_{fl}$和必须等价$\equiv_{mu}$的定义是一样的，即 $E \equiv_{fl} F$ 仅当 $E \equiv_{mu} F$。这是因为对于每一个可以从 E 开始执行的动作序列 σ 和每一个动作集合 Act 的子集 X，仅当 $\langle \sigma, X \rangle$ 不是从 E 开始的一个失败时，它才是一个必须对。

8.5.3 模拟等价

在如下的等价关系的定义中，我们隐式地假定一个给定的 agent 的集合 S 和一个动作集合 Act。当存在一个 agent 集合上的二元关系$\mathcal{R} \subseteq S \times S$，满足：

1. $E \mathcal{R} F$。

2. 如果 $E' \mathcal{R} F'$，且 $E' \xrightarrow{\alpha} E''$，那么存在某个 F''满足 $F' \xrightarrow{\alpha} F''$且 $E'' \mathcal{R} F''$。

我们说一个 agent F 模拟一个 agent E。根据这个定义，$\mathcal{R}$确立了 F 是如何模拟 E 的。假定 E 演化为 E'（通过执行某一动作序列），F 演化为 F'，E'和 F'通过$\mathcal{R}$相联系。那么，F'模拟 E'，即 $E' \mathcal{R} F'$。从 E'执行任意动作 α，得到某 agent E''可以通过从 F'执行 α 得到某 agent F''来模拟。那么，F''模拟 E''，即 $E'' \mathcal{R} F''$，且该定义可以如此反复应用到后面的 agent 上。

从模拟的定义可以得到一些直观的结论：如果 $E' \mathcal{R} F'$，那么

- 从 E'得到的任意动作序列也可以从 F'得到。
- 可以从 E'执行的所有动作也可以从 F'执行（因此，所有不可以从 F'执行的动作也不可以从 E'执行）。

如果 E 模拟 F 且 F 模拟 E，那么我们写成 $E \equiv_{sim} F$ 并且说 E 和 F 是模拟等价的。（形式化地，

我们首先需要证明$\equiv_{sim}$是一个等价关系。这可以从定义很简单地推出。）注意，确立$E\equiv_{sim}F$没有这样的要求：通过F模拟E的关系$\mathcal{R}$与通过E模拟F的一个关系$\mathcal{Q}$相一致。也就是说，$\mathcal{Q}=\mathcal{R}^{-1}$的情况不需要一定成立。

模拟等价比迹等价要好，因为通过重复地应用如上关系$\mathcal{R}$的定义，如果$E\equiv_{sim}F$，E的任意迹可以在F中被模拟，反之亦然。因此，E和F有同样的迹。如图 8.7 所示的例子展示了两个agent是迹等价，但不是模拟等价。根据这两个agent，某种选择会导致最终只有γ或只有δ被允许。然而，非确定性选择在两个 agent 中是在不同的地方实现的。在左边的 agent 中，这个选择是在一开始执行α时作出的；而在右边的 agent 中，选择是在执行动作β时作出的。

右边的 agent，$\alpha.(\beta.\gamma.Nil+\beta.\delta.Nil)$，模拟了左边的 agent，$\alpha.\beta.\gamma.Nil+\alpha.\beta.\delta.Nil$。这可以通过模拟关系

$$\mathcal{R}=\{(s_1,r_1),(s_2,r_2),(s_4,r_3),(s_6,r_5),(s_3,r_2),(s_5,r_4),(s_7,r_6)\}$$

来证明。另一方面，我们将展示左边的 agent 不能模拟右边的 agent。假定存在从由右边的 agent 演化而来的配置到由左边的 agent 演化而来的配置的模拟关系$\mathcal{Q}$，使得$r_1\,\mathcal{Q}\,s_1$。那么，因为α从s_1和r_1都是可执行的，我们必须有$r_2\,\mathcal{Q}\,s_2$或者$r_2\,\mathcal{Q}\,s_3$，或者二者都成立。我们将驳斥$r_2\,\mathcal{Q}\,s_2$，驳斥$r_2\,\mathcal{Q}\,s_3$可以类似地完成。假定$r_2\,\mathcal{Q}\,s_2$成立。因为β从r_2可以执行，所以我们必须有$r_3\,\mathcal{Q}\,s_4$和$r_4\,\mathcal{Q}\,s_4$。然而，$r_4\,\mathcal{Q}\,s_4$不能成立，因为只有δ可以从r_4执行，而只有γ可以从s_4执行，从而形成一个矛盾。

图 8.7 中的两个 agent 也可以被看成是失败等价。现在我们展示一个两个 agent 是模拟等价但不是失败等价的例子。图 8.8 展示了 agent $\alpha.\beta+\alpha$ 和 $\alpha.\beta$。容易看出在两个方向上存在模拟关系。从左边到右边，我们有关系$\mathcal{R}$使得$q_1\,\mathcal{R}\,l_1$，$q_2\,\mathcal{R}\,l_2$，$q_3\,\mathcal{R}\,l_2$和$q_4\,\mathcal{R}\,l_3$。q_3被l_2模拟并不是问题，因为从q_3可执行的动作的缺失并不意味着l_2必须没有可执行的动作。

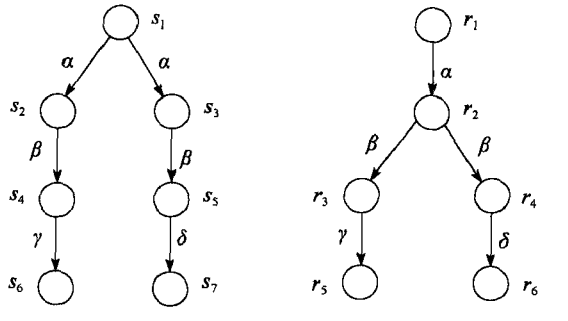


图 8.7　agent $\alpha.\beta.\gamma.Nil+\alpha.\beta.\delta.Nil$ 和 $\alpha.(\beta.\gamma.Nil+\beta.\delta)$

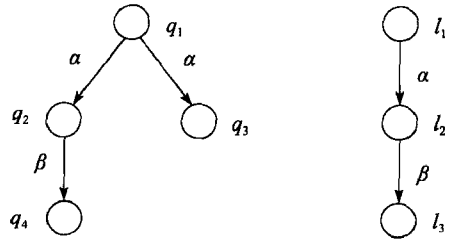


图 8.8　agent $\alpha.\beta+\alpha$ 和 $\alpha.\beta$

在另一个方向上，我们有模拟关系$\mathcal{Q}$使得$l_1\,\mathcal{Q}\,q_1$，$l_2\,\mathcal{Q}\,q_2$和$l_3\,\mathcal{Q}\,q_4$。注意q_3不能模拟l_2：动作β从l_2可执行但从q_3不可执行。模拟关系$\mathcal{R}$和$\mathcal{Q}$是唯一的，即它们不可以用其他在两个系统之间建立模拟的关系替代。这两个 agent 不是失败等价的：$\alpha.\beta+\alpha$有失败$\langle\alpha,\{\beta\}\rangle$，而$\alpha.\beta$没有。

练习 8.5.1　考虑如图 8.9 所示的两个 agent。用 CCS 表达式描述它们。证明左边的 agent 不能模拟右边的 agent。右边的 agent 可以模拟左边吗？

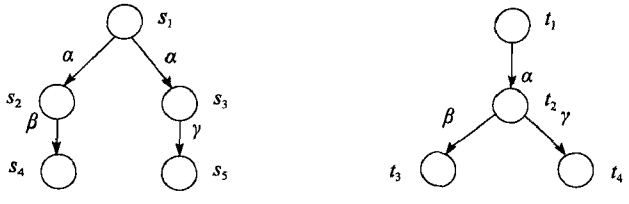


图 8.9　练习 8.5.1 中的两个 agent

8.5.4 互模拟和弱互模拟等价

模拟等价使用两个关系来定义：$\mathcal{R}$用右边的 agent 模拟左边的 agent，$\mathcal{Q}$用左边的 agent 模拟右边的 agent。回想一下，这些模拟关系不需要相一致，也就是不要满足$\mathcal{Q}=\mathcal{R}^{-1}$。例如，图 8.8 中的 agent 是模拟等价的，因为在两个方向上各存在一个模拟关系。然而，回想一下在这个例子当中，从左到右的唯一一个模拟与从右到左的模拟并不一致。特别地，$q_3\,\mathcal{R}\,l_2$，但是 $l_2\,\mathcal{Q}\,q_3$ 不成立。因此，不可能发现一个单一的关系能从两个方向上模拟。

形式化地，$E\equiv_{bis}F$ 当且仅当存在一个基于 agent 的集合 S 的二元关系$\mathcal{R}\subseteq S\times S$满足：

1. $E\,\mathcal{R}\,F$。

2. 对任意的 agent 对 E'和 F'，以及一个动作 α，如下两个条件成立：

   a) 如果 $E'\,\mathcal{R}\,F'$且 $E'\xrightarrow{\alpha}E''$，那么存在某个 F''使得 $F'\xrightarrow{\alpha}F''$和 $E''\,\mathcal{R}\,F''$成立。

   b) 如果 $E'\,\mathcal{R}\,F'$且 $F'\xrightarrow{\alpha}F''$，那么存在某个 E''使得 $E'\xrightarrow{\alpha}E''$和 $E''\,\mathcal{R}\,F''$成立。

从该定义紧接着可以得到，如果两个 agent 是互模拟，那么它们也是相似的；只要在模拟的两个方向上使用关系$\mathcal{R}$和$\mathcal{R}^{-1}$。在 8.5.3 节结尾处涉及图 8.8 中的 agent 间两个相互不一致的模拟关系的讨论，表明模拟和互模拟等价是不一样的。因此，互模拟等价是模拟等价的一种适当的精化。

我们来证明互模拟是失败等价的精化。令$\langle\sigma, X\rangle$为 E 的一个失败。我们将展示如果$E\equiv_{bis}F$，那么$\langle\sigma, X\rangle$也是 F 的一个失败。令$\mathcal{R}$为 E 和 F 之间互模拟的见证。令 E'为通过从 E 执行 σ 得到的 agent，使得 X 中的所有动作从 E 都是不可执行的。通过重复地使用互模拟的定义，我们可以证明存在一个从 F 执行 σ 得到的 agent F'，使得 $E'\,\mathcal{R}\,F'$成立。那么，X 中没有动作从 F'可执行。否则，必然存在一个 agent $F'\xrightarrow{\alpha}F''$，其中 $\alpha\in X$。那么必然存在一个状态 E''使得 $E'\xrightarrow{\alpha}E''$和 $E''\,\mathcal{R}\,F''$成立，与$\langle\sigma, X\rangle$为 E 的一个失败矛盾。因此，E 的每一个失败也是 F 的一个失败，通过对称的论证，F 的每一个失败也是 E 的一个失败。因此，E 和 F 是失败等价的。

图 8.7 中的两个 agent 为证明互模拟等价也是失败等价的适当的精化提供了证据。在 8.5.3 节中我们展示了这两个 agent 是失败等价但不是模拟等价的。因为互模拟等价的 agent 也是模拟等价的，因此这两个 agent 不可能是互模拟等价的。

不可见动作出现时，要求 agent 之间的互模拟作为标准可能太强了。它使得那些仅仅因不可见动作的执行次数不同而不同的 agent 都是可区别的。*弱互模拟*等价以类似于互模拟等价的方式定义，但是考虑到了经过扩展的事件关系。它允许在一个抽象的层次上推理被建模系统，且不对不可见动作被重复执行了多少次进行计数。形式化地，$E\equiv_{wbis}F$ 当存在一个 agent 间的关系$\mathcal{R}$，使得：

1. $E\,\mathcal{R}\,F$。

2. 对每一对 agent E'和 F'，以及 $\alpha\in Act\cup\{\varepsilon\}$，以下条件成立：

   a) 如果 $E'\,\mathcal{R}\,F'$且 $E'\overset{\alpha}{\Rightarrow}E''$，那么存在某个 F''使得 $F'\overset{\alpha}{\Rightarrow}F''$和 $E''\,\mathcal{R}\,F''$成立。

   b) 如果 $E'\,\mathcal{R}\,F'$且 $F'\overset{\alpha}{\Rightarrow}F''$，那么存在某个 E''使得 $E'\overset{\alpha}{\Rightarrow}E''$和 $E''\,\mathcal{R}\,F''$成立。

另一个互模拟定义的有趣的变体添加了条件 $E'\xrightarrow{\tau}$当且仅当 $F\xrightarrow{\tau}$。这意味着对于等价的 agent，要么都可以通过执行无数多次 τ 动作实现发散，要么都不可以发散。

8.6 等价关系的层级

如图 8.10 所示为将要讨论的等价关系的层级。包含更多等价关系的复杂层级，参见 [53]。从一个等价关系到另外一个的边表示前者是后者的一个适当的精化。也就是，任意两个在后一个关系中等价的 agent 在前一个关系中必定也是等价的；更进一步，存在一对 agent 根据后一个

关系等价但根据前一个关系不等价。换句话说，前一个等价关系强于（蕴含）后一个关系。在[54]中，基于扩展的事件关系（如前面提到的弱互模拟）的等价被加入到了层级当中。

有趣的是，这个层级强依赖于动作可能是非确定性的这一事实。也就是，从同一个 agent 执行某一相同的动作可能产生超过一个后继 agent，如在 $a.\beta+a.\gamma$ 中。如果这种非确定性的动作不被允许，那么上面所述的所有等价彼此都一样。事实上，人们可以用语言等价这一简单标准代替。注意，不允许非确定性的动作并不意味着同时废除非确定性：比如，在 $\alpha+\beta$ 中系统仍然可以非确定性地决定执行动作 α 还是另一个动作 β。

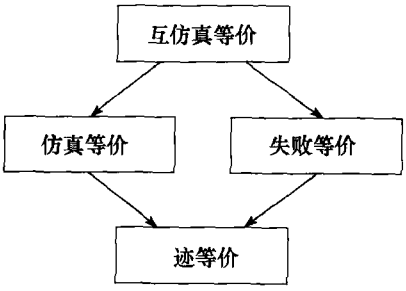


图 8.10　等价关系的层级

8.7　用进程代数研究并发

进程代数的框架允许我们研究一些不同构造的属性，例如非确定性选择和并发的结合。例如，我们可以证明如下关于互模拟关系的等价：

交换率　$A+B\equiv_{bis}B+A$，$A\parallel B\equiv_{bis}B\parallel A$。

结合律　$A+(B+C)\equiv_{bis}(A+B)+C$，$A\parallel(B\parallel C)\equiv_{bis}(A\parallel B)\parallel C$。

非确定性选择的幂等律　$A+A\equiv_{bis}A$。

更弱的等价关系包含更强的等价关系的所有等价，并且可能有附加的等价。例如，对于弱互模拟关系，除以上的等价外，我们有

$$\tau.A\equiv_{wbis}A \tag{8.8}$$

使用进程代数研究的一个有趣的现象是全等（congruence）的概念。一个全等是一个等价关系同样允许在任意上下文中替换。令 A，B，C 为三个 agent。通过将 A 中所有的 B 替换为 C 得到的 agent 表示为 $A\{C/B\}$。例如，A 为 $\alpha+\beta$，B 为 β，C 为 $\tau.\beta$。那么 $A\{C/B\}$ 为 $\alpha+\tau.\beta$。agent 之间的全等关系要求满足用一个全等于 agent B 的 agent C 来替代 B 从而产生一个全等于 A 的 agent。形式化地：如果“$\cong$”是一个 agent 之间的全等关系，且 $B\cong C$，那么 $A\cong A\{C/B\}$。

对 agent 的结构应用归纳法，可以证明，互模拟等价是一个全等（参见[100]）。另一方面，弱互模拟等价不是一个全等。根据式（8.8），可以观察到：

$$\beta\equiv_{wbis}\tau.\beta \tag{8.9}$$

但是，很容易证明

$$\alpha+\beta\not\equiv_{wbis}\alpha+\tau.\beta \tag{8.10}$$

我们可以对式（8.10）中右边的 agent 应用扩展事件 ε，得到一个只有 β 被允许的 agent。对左边的 agent 应用扩展事件 ε 没有导致变化，允许 α 首先执行。因此，设置 $A=\alpha+\beta$，$B=\beta$，$C=\tau.\beta$ 为证明弱互模拟关系不是全等提供了一个反例[100]。

使用进程代数可以展示的另一个有趣的现象是在处理动作精化时交错语义的缺乏。注意并发组合规则将交错语义强加于并发组合。表达式

$$E=\alpha\parallel\beta.\gamma \tag{8.11}$$

在执行 α 后可以演化为 $0\parallel\beta.\gamma$，或者在执行 β 后演化为 $\alpha\parallel\gamma$。因此我们可以说 E 等价（在互模拟等价下）于表达式

$$E'=\alpha.\beta.\gamma+\beta.\alpha.\gamma+\beta.\gamma.\alpha \tag{8.12}$$

因此允许的执行序列为 $\alpha\beta\gamma$，$\beta\alpha\gamma$，$\beta\gamma\alpha$，交错了左边进程的动作 α 和 β 以及接着的右边进程中的 γ。

现在考虑动作精化的操作，其中在一个 agent 中一个动作可以被一个有限动作序列所替代（使用“.”运算符隔开每个动作）。例如，动作精化可以用于描述一个系统的逐步精化。我们现

在可以用进程代数展示在动作精化中交错语义是不封闭的。考虑 agent：

$$F = a \parallel \delta \tag{8.13}$$

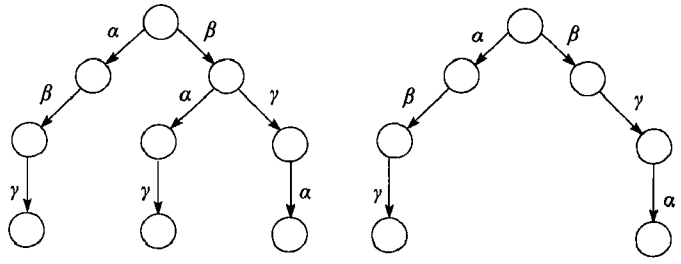
它允许动作 a 和 δ 并发地执行。因此，根据 CCS 的语义，以任意的交错顺序。再一次，我们将并发组合转换为一个等价的非确定性 agent：

$$F' = a.\delta + \delta.a \tag{8.14}$$

现在，如果我们精化 F 式（8.13)）中的 δ，将其替换为序列 $\beta.\gamma$，从式（8.11）我们得到 agent E（等价于式（8.12）中的 E')。但是，如果我们对式（8.14）中等价于 E 的 agent F' 应用同样的精化，我们得到

$$C = a.\beta.\gamma + \beta.\gamma.a \tag{8.15}$$

后面的 agent C 不等价于（在本章提出的任意等价下）E，因为它不允许序列 $\beta a \gamma$，即交错 a 于 β 和 γ 之间，如图 8.11 所示。



agent E（或 E'）　　agent C

图 8.11　agent E 和 C

例子

在接下来的例子中 [18]，使用两种不同方法定义一个容量为 2 的队列，并对两者之间的关系进行研究。第一个定义如下：

$$Empty2 \triangleq put.Half2$$

$$Half2 \triangleq put.Full2 + get.Empty2$$

$$Full2 \triangleq get.Half2$$

这个序列的状态空间如图 8.12 的左边所示。

第二个定义是非直接的。我们首先定义一个缓冲队列如下：

$$Empty1 \triangleq put.Full1$$

$$Full1 \triangleq get.Empty1$$

这个序列的状态空间如图 8.12 的中间所示。我们创建容量为 1 的队列的两个副本，并将它们连接在一起。为此，我们重命名（通过重新标记）第一个队列中的 get 动作为 $link$，重命名第二个副本中的 put 动作为 $\overline{link}$。然后我们用并发运算符 “$\parallel$” 将两个副本合并。$link$ 和 $\overline{link}$ 的同步发生所表示的第一个副本的 get 和第二个副本的 put 的同步，将被不可见动作 τ 所替代。为了防止不同步的出现，我们使用如下隐藏运算符。

$$B1 \triangleq Empty1[link/get]$$

$$B2 \triangleq Empty1[\overline{link}/put]$$

$$Queue2 \triangleq (B1 \parallel B2) \setminus \{link\}$$

$Queue2$ 的状态空间如图 8.12 的右边所示。

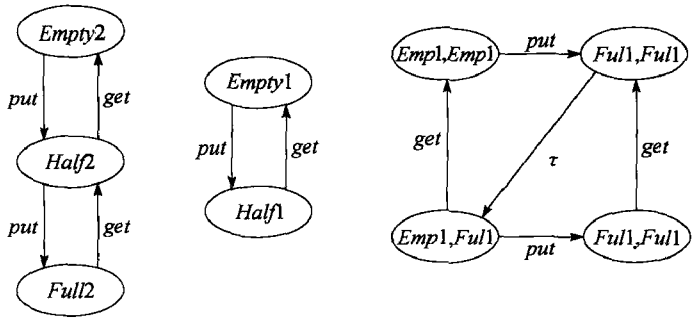


图 8.12　队列的进程代数模型

根据上述等价的定义，可以证明两位序列的这两种实现（*Empty2* 和 *Queue2*）不是互模拟的。特别是，*Queue2* 包含不可见动作，而 *Empty2* 不包含。但是，根据弱互模拟定义，这两个 agent 是等价的。

8.8　计算互模拟等价

考虑有限状态 agent 的情形。如果互模拟等价被用于比较一个实现与一个设计的正确性准则，那么一个检验它的有效方法是必需的。

我们给出一个检查 agent E 和 F 之间的互模拟等价的经典算法。开始我们构造状态空间 $\langle S, \Delta\rangle$，其中 S 是可以从 E 或者 F 演化得到的 agent 的有限集合（因此，$E, F \in S$），Δ 为 agent 之间的转换关系，即 $\Delta \subseteq S \times (Act \cup \{\tau\}) \times S$。该算法重复地将 S 的状态划分至不相交的子集中。它从一个包含 S 中所有节点的子集开始，精化划分，即分解节点的子集为更小的子集，直到每一个子集 $P \in \mathcal{P}$ 内包含的 agent 彼此之间互模拟等价。就是说，对于每个 $G_1, G_2 \in P$，$G_1 \mathcal{R} G_2$。

通常在 (S, Δ) 上有超过一种方式满足互模拟关系的定义，即有不止一个关系 $\mathcal{R} \subseteq S \times S$ 满足互模拟等价的定义。一个简单的例子为一个关联每个 agent 只到自身的等价关系。当然这不是一个有趣的等价关系。总有一个最大的（即最粗粒度的）这种关系，即一个包含划分了 S 的最大的这种互模拟等价类。我们对找到在互模拟的定义中用到的最大的关系 $\mathcal{R}$ 感兴趣。然后我们可以检验 E 和 F 是否在划分的相同子集中。

划分算法如下：

1. 创建初始划分 $\mathcal{P} \in \{S\}$。

2. 重复直至不再有变化：

找到在划分 $\mathcal{P}$ 中是否存在两个（不一定要不同）元素 T_1，T_2（即 agent 的两个集合）和一个动作 $\alpha \in Act$ 使得以下成立：可以将 agent 的集合 T_1 分解为两个非空的且不相交的子集 S_1，S_2 满足

- 对每个 agent $E \in S_1$，存在一个 agent $E' \in T_2$ 满足 $E \xrightarrow{\alpha} E'$。
- 不存在 agent $E \in S_2$ 使得对某个 agent $E' \in T_2$，$E \xrightarrow{\alpha} E'$ 成立。

如果存在这种子集，用子集 S_1 和 S_2 替换 $\mathcal{P}$ 中的 T_1。

根据这个算法，区别 agent 从而把它们放到不同的子集中去的一个方法是，当某 agent 存在一个标记了动作 α 的事件，而其他 agent 没有标记了 α 的事件。另一种划分 agent 的方法是，两个 agent 都有标记了 α 的事件，但却分别演化成已经划分在不同子集中的 agent。

通过对算法执行的任意阶段中划分的数目进行简单地归纳，我们可以看到：如果一个 agent 的集合被分解成两个子集，那么我们可以通过这些 agent 不可能是互模拟的来区分它们。反过

来，我们可以对划分的长度进行归纳，从而证明：在算法的任意阶段，(S, Δ) 上的任意互模拟是当前划分的一个精化（特别地，包含最终的划分）。这意味着通过算法得到的互模拟是最大的那个。划分的数目以 S 的大小为边界，因为每个子集必须至少包含一个 agent。如果 n 是 S 中 agent的数目，m 是 Δ 中动作的数目，那么时间复杂度为$O(n\times m)$。

考虑两个 agent $A \triangleq \alpha.((\beta)+(\gamma.\delta.A))$ 和 $B \triangleq \alpha.\beta+\alpha.\gamma.\delta.B$。我们在图 8.13 中给出它们的图。每个节点对应于上述的每个 agent 或者从它演化得到的。节点 s_0 对应于 $A \triangleq \alpha.((\beta)+(\gamma.\delta.A))$，节点 r_0 对应于 $B \triangleq \alpha.\beta+\alpha.\gamma.\delta.B$。图中的边对应于事件而节点对应于 agent。例如，s_1 由 s_0 通过执行 α 动作而得到，因此 s_1 对应于$(\beta)+(\gamma.\delta.A)$。$r_0$ 到 r_1 的边对应于 B 中左边的 α 选择。因此，r_1 对应于 agent β。r_0 到 r_4 的边对应于在 B 中选择右边的 α 并得到 agent $\gamma.\delta.A$。

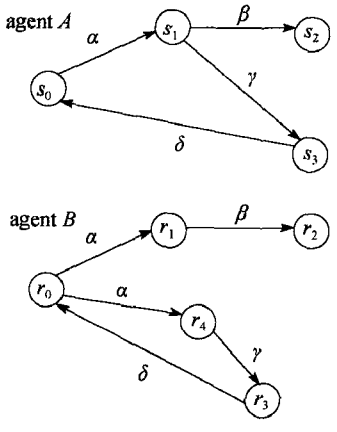


图 8.13 agent A 和 B 的状态空间

我们通过将所有节点放入一个集合来开始该算法：

$$S = \{s_0, s_1, s_2, s_3, r_0, r_1, r_2, r_3, r_4\}$$

类似地，我们将 Δ 设置为两个状态图中转换的集合：

$$\Delta = \{s_0 \xrightarrow{\alpha} s_1, s_1 \xrightarrow{\beta} s_2, s_1 \xrightarrow{\gamma} s_3, s_3 \xrightarrow{\delta} s_0, r_0 \xrightarrow{\alpha} r_1,$$
$$r_0 \xrightarrow{\alpha} r_4, r_1 \xrightarrow{\beta} r_2, r_4 \xrightarrow{\gamma} r_3, r_3 \xrightarrow{\delta} r_0\}$$

我们第一次根据动作 α 作出划分的选择。只有节点 s_0 和 r_0 可以执行 α。因此，我们得到如下两个子集的划分：

$$\{s_0, r_0\}$$
$$\{s_1, s_2, s_3, r_1, r_2, r_3, r_4\}$$

我们根据动作 β 划分第二个子集，β 仅可被节点 s_1 和 r_1 执行，得到

$$\{s_0, r_0\}$$
$$\{s_1, r_1\}$$
$$\{s_2, s_3, r_2, r_3, r_4\}$$

接着，我们根据动作 γ 划分第二个子集。节点 s_1 可以执行一个 γ 动作，而 r_1 不行。

$$\{s_0, r_0\}$$
$$\{s_1\}$$
$$\{r_1\}$$
$$\{s_2, s_3, r_2, r_3, r_4\}$$

现在，我们根据 γ 划分第四个集合。这个集合中唯一可以执行动作 γ 的节点是 r_4。因此它成为一个一元的集合，与其他分开。

$$\{s_0, r_0\}$$
$$\{s_1\}$$
$$\{r_1\}$$
$$\{s_2, s_3, r_2, r_3\}$$
$$\{r_4\}$$

接着我们根据 δ 划分第四个集合。节点 s_2 和 r_2 不能执行动作 δ，而 s_3 和 r_3 可以。

$$\{s_0, r_0\}$$
$$\{s_1\}$$
$$\{r_1\}$$

$\{s_2, r_2\}$

$\{s_3, r_3\}$

$\{r_4\}$

现在我们根据 a 划分第一个集合。s_0 和 r_0 都能执行动作 a。但是，从 s_0 执行 a 得到状态 s_1，从 r_0 执行 a 得到状态 r_1。因为这两个状态属于划分中的不同子集，我们需要将 s_0 和 r_0 分离。

$\{s_0\}$

$\{r_0\}$

$\{s_1\}$

$\{r_1\}$

$\{s_2, r_2\}$

$\{s_3, r_3\}$

$\{r_4\}$

到目前为止我们知道 A 和 B 不是互模拟的，因为它们分别用状态 s_0 和 r_0 表示，属于划分中不同的子集。如果我们想继续划分进程，可以根据 δ 对第六个子集进行划分，分离 s_3 和 r_3。理由是从 s_3 我们到达 s_0，从 r_3 我们到达 r_0，二者现在处于不同的子集当中。

$\{s_0\}$

$\{r_0\}$

$\{s_1\}$

$\{r_1\}$

$\{s_2, r_2\}$

$\{s_3\}$

$\{r_3\}$

$\{r_4\}$

我们不可以再进一步划分了：只有一个多于一个节点的子集，它包括了 s_2 和 r_2。这些节点是不可以分解的，因为对它们每一个而言，没有允许执行的动作。

检查有限状态的 agent E 和 F 的弱互模拟等价可以从计算 E 和 F 的状态空间开始。然后计算扩展的动作关系。这通过类似于计算一个关系的*传递闭包*的算法完成。这样的算法，例如 Floyd 和 Warshall 算法 [145]，可以在图大小的立方时间内完成（即$\mathcal{O}(m^3)$）。

这些复杂度以关于被比较的状态空间大小的函数的形式给出。正如 6.9 节中讨论的，测量输入大小是相当有偏见的方法，因为状态空间可能在指数级别上大于系统的进程代数描述。一个更加有效率的算法，其时间复杂度为$\mathcal{O}(m \times \log n)$，在 [111] 中给出。

8.9　LOTOS

LOTOS(Language Of Temporal Ordering Specification)是一个进程代数标准，由 ISO（International Standard Organization，国际标准化组织）颁布 [72]。它包括进程代数 CCS，一种带数据类型的代数化的规范 [44]。在这一节中我们将描述 LOTOS 的主要特征并将其与 CCS 比较。要了解更详细的 LOTOS 的描述，参见 [17]。

在 LOTOS 中，agent 被称为*进程*。一个进程定义如下：

process *process_name* [*actions_list*] :=*behavior_expression* end proc

其中的动作列表用逗号分隔。与 CCS 不同，LOTOS 不包含互补动作。

LOTOS *behavior_expression* 中的某些运算符在 CCS 中有直接对应，尽管表示方法通常有小

小的不同。LOTOS中动作的前缀运算符为“;”，而不是“.”。非确定性选择为“[]”，替代了“+”。不可见动作为i，替代了τ。限制在LOTOS中表示为“\”，与在CCS中一样；但是，动作列表在LOTOS中以方括号包围，“[”和“]”，而不是大括号。终止的表示为exit而不是0。exit的执行结果为特殊动作δ。

LOTOS提供了三种合并并发进程的方法：

- 全同步：用运算符“‖”表示，允许所包含的进程仅可以在执行相同的动作时演化。因此$a;B \parallel a;C$可以通过执行a演化为$B \parallel C$，而$a;B \parallel d;C$的两个并发组件都不能继续。
- 纯交错：用运算符“|||”表示，允许从任意一个并发进程选择一个动作执行，而其余的进程保持不变。因此，$a;B|||d;C$可以通过执行a演化为$B|||d;C$，或者通过执行d演化为$a;B|||C$。在纯交错组合之下，在相同动作的不同发生之间没有同步。因此，进程$a;B|||a;C$可以演化为$B|||a;C$或者$a;B|||C$，但是与CCS中的并发组合不同，它不能演化为$B|||C$。
- 选择性同步：表示为以下语法

$$process \mid [action\_list] \mid process$$

左边和右边的进程可以通过列表中的动作同步，或通过其他动作交错。因此，$a;B|[a]|a;C$可以通过执行a演化为$B \parallel C$。进程$a;B|[a]|d;a;C$只能通过执行d演化为$a;B|[a]|a;C$。它不可以通过执行a演化，因为两个并发进程之间的同步依赖于a，而a只被其中一个进程所允许。但是，在执行d之后，两个进程可以通过执行a同步。

LOTOS有一些在CCS中不能直接形成的结构。使能（enabling）运算符“≫”允许顺序组合。因此，$A \gg B$像A一样行动，之后如果A成功终止，即以exit结束，则它像B一样行动。在A和B的执行之间有不可见动作i的一次发生，代替了在A结束时的exit所对应的特殊动作δ。中断（disruption）运算符“[>”允许设定中断。因此，$A [> B$开始像A一样行动。在A执行的任意时刻，它可能被中断，然后B的执行开始。如果A成功终止，即执行到exit，那么B不可以再执行。如果A不能成功终止，那么B必须执行。hide（隐藏）运算符有如下语法：

$$\text{hide } [action\_list] \text{ in } process$$

它用不可见动作i替换*action_list*中的动作。隐藏和限制被用于禁止与进程的某些动作的交互。但是，限制不允许含有被限制动作的执行，而隐藏允许它们执行并随后将它们替换为不可见动作。

8.10 进程代数工具

concurrency workbench可以从爱丁堡大学获得，使用如下URL：

http://www.dcs.ed.ac.uk/home/cwb

FC2Tools包可以从INRIA获得，通过如下URL：

http://www-sop.inria.fr/meije/verification

PSF工具包可以从阿姆斯特丹大学获得，通过如下URL：

http://adam.wins.uva.nl/~psf

注意：形式化方法系统的使用通常要求填写并发送一份恰当的发布表格，确认遵守某些使用条款。

8.11 扩展阅读

Robin Milner所著的关于进程代数的经典书籍包括：

R. Milner, *Communication and Concurrency*, Prentice-Hall, 1995.
R. Milner, *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press, 1999.

最近的进程代数的一个扩展，允许*移动性*（mobility），即通信拓扑的动态变化，是 π 演算。上述列表中的第二本书有对 CCS 和 π 演算的描述。

近期关于通信协议，包含对 LOTOS 的描述以及其他 ISO 标准（如 SDL 和 ESTELLE）的书是：

R. Lai, A. Jirachiefpattana, *Communication Protocol Specification and Verification*, Kluwer Academic Publishers, 1998.

其他关于进程代数的书籍包括：

J. C. M. Baeten, W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1990.
J. C. M. Baeten, ed., *Applications of Process Algebra* Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 1991.
G. Bruns, *Distributed System Analysis with CCS*, Prentice-Hall, 1996.
M. Hennessy, *Algebraic Theory of Processes*, MIT Press, 1988.

关于不同的进程代数的扩展和它们之间的比较可以在以下两篇论文中找到：

R. J. van Glabbeek, The Linear Time-Branching Time Spectrum (Extended Abstract), CONCUR 1990, Theories of Concurrency, Amsterdam, The Netherlands, Lecture Notes in Computer Science 458, Springer-Verlag, 1990, 278–297.
R. J. van Glabbeek, The Linear Time–Branching Time Spectrum II, E. Best (ed.), 4th international conference on Concurrency theory, CONCUR 1993, Lecture Notes in Computer Science 715, Springer-Verlag, Hildesheim, Germany, 66–81.

第9章

Software Reliability Methods

软件测试

……瓶颈上系着一个纸标签，上面漂亮地印着两个大字："喝我"。说"喝我"当然挺好，可是聪明的小爱丽丝是不会急于这么干的。"不行，我得先看看，"她说，"上面是不是写着'有毒'。"

刘易斯·卡洛尔《爱丽丝漫游奇境记》

测试是基于给定的准则对系统的执行进行抽样的一个过程。测试过程将系统的每次执行与规约进行比较，并将其中的不一致作为错误报告。由于测试通常是对系统执行的抽样，而不是检查所有的系统执行，所以并不能保证所有的错误都能被覆盖到。测试方法究不能像前面几章中所介绍的形式化方法那样全面地对系统的正确性提供保证。因此，一些研究人员甚至不将测试归为形式化方法。然而，当给定系统规模较大，不能进行人工或自动的验证时，测试往往能提供一个较为实用的解决方案。事实上，测试基本上是提高软件质量的最为广泛使用的方法。

显而易见，即使是最严格的测试技术也无法保证经过测试后的系统在任意环境下都能正确运行。例如，测试过程中可能忽略了一些内部参数，如温度、湿度等。对一个自动售货机出售巧克力棒的功能连续测试10次，每次投入正确数量的钱币，按了正确的按钮后，取货口均成功送出了巧克力棒，并不能保证第11次也能正确运行。前面所进行的测试仅仅是增加了对系统正确执行的期望值。虽然下面这种情况出现的可能性不大，但是该自动售货机仍有可能在下午5点前是正常工作的，5点后取货口送出的却是爆米花而不是所购买的巧克力棒。尽管如此，使用前面几章中所描述的方法验证系统的设计，并对实际系统的行为正确性进行测试，肯定能减少系统出现错误和异常行为的可能性。

测试并不能用于证明待测程序中没有错误，也不能保证程序会正确地完成预期目标。同样，测试也并不能保证找到程序中所有的（甚至是一些）错误，不存在一个充分的测试用例集可以保证做到这点。测试仅仅是一个在指定的条件和参数下运行待测程序，并试图找出错误的过程。

在某种意义上，测试员的目标是处于程序员的对立面的：一个成功的测试员需要发现程序中存在的错误。因此，至少在理论上，程序员不应该对自己所编写的代码进行测试。然而在实践中，经常会要求程序员对自己所编写的代码进行测试。

尽管测试并不能保证会发现给定程序中的所有错误，但测试易于进行，并能以合理的开销提高系统的可靠性。特别是与演绎验证相比，测试所需要的时间开销和人力资源均明显要少。当验证难于实施，模型检验不可行时（例如，在出现无限的或巨大的状态空间，或者复杂的数据结构时），测试方法往往仍然是适用的。

软件测试包括下面几个层次和阶段：

单元（模块）测试。最底层的测试阶段，对代码的小片段单独进行测试。

集成测试。对多个代码片段协同进行测试，这些代码片段可能是由不同团队开发的。

系统测试。将系统作为一个整体进行测试，通常用于审查软件的功能性。

验收测试。通常由用户进行，检查所开发的系统是否满足其需求。

回归测试。在维护阶段进行，当对系统中的部分模块进行修改、校正或者升级时，检查系统各项功能是否仍能正确运行。回归测试通常在为已测试系统增加新功能时使用。在这种情况下，

可以重复使用以往的测试用例来检查原有的各项功能是否能正确运行。

压力测试。在极端条件下对系统进行测试，例如，在不正常的有大量用户或大量数据的情况下。

*白盒测试*是一种以审查系统内部细节（如代码）为基础来对系统进行排错的技术。部分测试从业者并不喜欢使用*白盒*（white box）这个术语，而更倾向于使用直观的*透明盒*（transparent box）这个术语来强调测试过程中代码的可见性这一特征。*执行路径*由一个待测代码中出现的控制点和指令的序列组成。我们可以将一条执行路径视为待测代码流程图中的一条路径。待测程序执行路径的数量可能相当巨大，甚至是无限的。由于程序路径数量巨大，遍历所有路径往往是不可行的。为解决这一问题，研究人员提出了多种测试覆盖准则。根据这些覆盖准则，仅需检查相对较少的一些路径，就能达到较高的错误发现率。*代码覆盖分析*可以用于对测试的覆盖度进行定性和定量的度量。在测试过程结束后，它可以对代码中仍然存在错误的可能性进行预测。

与白盒测试不同，*黑盒测试*不是基于系统内部结构的。在某些情况下，系统的内部结构可能是不可获知的，黑盒测试使用待测系统的已知接口所允许的实验进行测试。在某些情况下，即使系统的内部结构部分或全部可知时仍会使用黑盒测试，以避免具体的实现方式给测试带来影响和偏差。

由于白盒测试直接处理代码，因此它更适用于较低层次的测试，如单元测试和集成测试。黑盒测试则适合检查系统的功能性，因此更适用于系统测试和验收测试。当然，黑盒测试也可用于单元测试。

在单元测试和集成测试阶段找到错误可以帮助定位错误所在的位置，而较高层次的测试仅能对错误的大致位置给出提示。在软件生命周期的较早阶段找出错误并立即修复所需的成本更低。另外，较低层次的测试，如单元测试，允许对软件的多个部分同时进行开发和测试。

测试基于在待测系统上自动或手工进行的一系列*实验*（experiment）。测试人员通过使用一些测试方法（可能还会使用某些测试工具）来生成一组测试用例，得到一个*测试套件*（test suite）。测试的执行包括与待测系统间的交互以及逐个执行测试套件中的测试用例。测试环境必须允许测试人员逐步运行实验，并进行对其观察。获取一个合适的测试环境可能需要在待测软件中增加一些代码。实际的测试过程还可能包括为程序在预先指定的位置提供用户输入，并在运行结束后观察所输出的结果。在其他情况下，测试可能强制程序从某个特殊的状态开始运行，或者当程序执行到某个特殊位置时对某内部变量的值进行比较。为完成实验和观察，需要增加一些特殊的代码来运行测试。

接下来我们首先介绍几种白盒测试技术，然后从9.9节开始介绍黑盒测试技术。

9.1 审查和走查

审查（inspection）和*走查*（walkthrough）都是人工的测试方法，一般在在会议中由一小组人员进行，通常需要1～2小时。实验数据表明，在同行审查会议中通常可以发现代码中30%～70%的错误。

在*代码审查*过程中，团队的一个成员是*协调员*（moderator）。他负责分发材料，控制会议进度并记录错误。测试过程主要是程序员逐行解释待审查代码。团队成员从一个潜在错误的完整列表中选取一些问题进行提问和讨论。下面是一些与典型程序错误相关的这类问题的示例：

是否所有的变量都在使用前被初始化？

数组变量的下标是否越界？

是否所有的变量都已声明？

过程调用及对应的过程声明的参数数量和类型是否一致？

是否有除零操作？

下面是对代码审查的质量有影响的几个因素。

- 代码审查的准备时期至关重要。如果代码是全新的，可能需要程序员准备一个简短的讲座对代码进行介绍。在这种情况下，可能更适于称此为代码走查。
- 为实现高质量头脑风暴会议，代码审查员的数量、程序员和外部审查员的混合很重要。
- 协调员要确保审查过程中发现的重要问题都得到了记录，并分配合适的人选负责解决审查中所发现的问题。
- 发现错误的数量可以用做一个度量来评估审查成功的程度。对不同类型的系统，有统计数据表明典型的错误数量。

*代码走查*同样由一个比较小的团队展开，与代码审查时的安排相似。同样，协调员控制会议进度并记录错误。测试人员通过前面构造的测试用例模拟计算机的行为。测试用例使用标准文档进行描述。

上述两种测试方法能否获得成功依赖于团队成员的准备和协作。协调员应该设定会议时间并提前向团队成员分发待测代码。团队其他人员则需要在会议前熟悉代码。在审查和走查这样人与人之间的交互活动中，一些心理因素至关重要。例如会议的长度（在类似于头脑风暴的会议中，这通常会直接影响到注意力集中的程度），对团队成员相互矛盾目标的控制能力（比如，一方面要找出程序中的错误，另一方面又会为对代码的批评进行辩护）。

9.2 控制流覆盖准则

在单元测试中，一个测试用例通常对应于所选择的一条执行路径。路径可以进行选择，比如可以通过指定初始值和执行过程中所需要的输入来进行选择。（然而，在程序可能出现非确定性行为时，比如存在并发的时候，指定初始值和输入不足以完全控制整个执行过程。）测试人员可以执行一条路径并将输出结果与预期的输出进行比较。测试过程中，假定测试人员对待测系统正确的行为是已知的，并能发现待测程序实际行为的偏差。

在测试过程中，很少会采用完整地检查系统所有执行的方式。因此，测试通常是基于特定的*覆盖准则*进行的。按照一定的覆盖准则，可以将可能会发现同样错误的执行归为一个集合。（在实践过程中，测试发现的可能是引发错误的缺陷源，而并不总是错误。）然后测试人员在每一个集合中抽取一个作为执行样本。例如，可以将沿着待测程序对应的流程图中同一条路径运行的执行归为一个集合。

总而言之，集合数量越大，每个集合所包含的测试用例的数量就越小。通常（但并不一定）一个覆盖准则需要检查的执行路径越多，发现程序中潜在错误的可能性就越高，但完成测试所需要的工作量也越多。覆盖准则可以被视为检验代码的启发式方法，其中一些覆盖准则将在下面介绍。

接下来我们将介绍几种主要的覆盖准则并通过一个示例来说明它们之间的区别。图 9.1 是示例程序的一部分所对应的流程图。该示例不需要用户提供输入。我们将基于所介绍的各种覆盖准则，给出针对该流程图的一些测试用例的示例。

为更好地使用这个特定的例子来强调覆盖准则的不同，在测试用例中使用在判定谓词（decision predicate）$x \equiv y \wedge z > w$ 之前（即刚刚递增了 y 之后）的变量状态来描述。但在一般情况下，在路径入口处采用赋值表达式来描述测试用例更为合理，因为我们可以适当地对测试进行初始化。将测试用例的规约向路径入口处传播的方法将在 9.4 节中介绍。在随后的章节中，

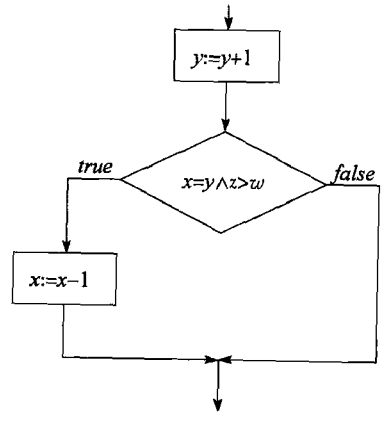


图 9.1 部分流程图

我们使用条件（condition）这个术语来表示一个简单的一阶公式（参见3.3节），即一阶项关系的应用。

9.2.1 语句覆盖

程序中的每条可执行语句（例如，赋值、输入、判定、输出）在至少一个测试用例中出现。

对语句覆盖准则，我们使用下面的赋值使判定谓词 $x \equiv y$ 和 $z > w$ 的计算值均为 TRUE：

$$\{x \mapsto 2, y \mapsto 2, z \mapsto 4, w \mapsto 3\} \tag{9.1}$$

值得注意的是这样不能覆盖到判定谓词的值为 FALSE 的情况。因此，可能会缺失某些选择 *false* 边的情况所需的必要的计算，但未在代码中出现。

9.2.2 边覆盖

流程图中的每条可执行边都在某测试用例中出现。

特别地，在此覆盖准则下，需要覆盖待测程序所有判定谓词的 *true* 和 *false* 分支（例如 if-then-else 命令，或者 while 循环）。边覆盖准则也常被称为*分支覆盖*或者*判定覆盖*。

为满足边覆盖准则，我们需要在测试用例（9.1）的基础上增加另一个测试用例，用以检查谓词的判定值为 FALSE 的情况，使该测试用例在判定节点选择 *false* 边。增加的测试用例为：

$$\{x \mapsto 3, y \mapsto 3, z \mapsto 5, w \mapsto 7\} \tag{9.2}$$

测试用例（9.1）和（9.2）将判定谓词 $x \equiv y \wedge z > w$ 作为一个单独的单元来覆盖，而并未作为两个独立的条件来考虑。后一个测试用例中选择 *false* 出边是因为 $z > w$ 不能被满足。但条件 $x \equiv y$ 可能是错误的，例如应该是 $x \geqslant y$，当 $x > y$ 时，递减 x 的语句仍然应该被执行到，而这种情况目前未能被测试到。

在对布尔运算符使用“短路”（short circuiting）规则的程序设计语言中，将判定谓词作为一个整体来测试会带来问题。在该规则中，形如 $A \wedge B$ 的表达式，当 A 的值为 FALSE 时，整个表达式的值将判定为 FALSE，而不需要求 B 的值。类似地，形如 $A \vee B$ 的表达式，当 A 的值为 TRUE 时，整个表达式的值将判定为 TRUE，而不需要求 B 的值。以判定谓词 $A \vee (B \wedge C)$ 为例，其中 C 是有副作用的函数调用，函数的返回值为一个布尔值。当 A 的值为 TRUE，或 A 和 B 的值均为 FALSE 时，该判定谓词的两个分支均被执行到。但在上述两种情况下，C 都没有被执行到。

边覆盖准则的另一个缺点是仅考虑了条件的布尔值，如条件 $x \equiv y$ 可以通过不同的方式置为假，增加 $x > y$ 的测试用例并不能反映 $x < y$ 的情况。要解决最少运算规则所引发的问题，则需要更加严格的覆盖准则，如边/条件覆盖或组合覆盖。

9.2.3 条件覆盖

每个判定谓词都是简单的一阶布尔表达式的组合。它包括两个表达式的比较，或某些程序变量间关系的应用。如果一个可执行条件的值既可以是 TRUE 也可以是 FALSE，那么它在某些测试用例中被计算为 TRUE，在另一些测试用例中则被计算为 FALSE。

对条件覆盖，我们使用测试用例（9.2）来使 $x \equiv y$ 的值为 TRUE 且 $z > w$ 的值为 FALSE。我们为 $x \equiv y$ 的值为 FALSE 且 $z > w$ 的值为 TRUE 的情况增加下面的测试用例：

$$\{x \mapsto 3, y \mapsto 4, z \mapsto 7, w \mapsto 5\} \tag{9.3}$$

值得注意的是在测试用例（9.2）和（9.3）中，整个判定谓词的值均为 FALSE，这将会缺少对递减操作语句的检查。

9.2.4 边/条件覆盖

此覆盖准则要求所有可执行边和条件都被覆盖到。

按照边/条件覆盖准则，我们选择测试用例（9.1）、（9.2）和（9.3）。这将保证判定谓词能取到真和假的输出，而且每个独立的条件，即 $x \equiv y$ 和 $z > w$，都将会被分别判定为 TRUE 和 FALSE。

9.2.5 条件组合覆盖

条件组合覆盖与条件覆盖相似，但条件组合覆盖考虑的不是每个单独的条件，我们要求每条判定谓词的所有布尔值组合方式都必须出现在某个测试用例中。

因此，形如 $A \wedge (B \vee C)$ 的判定谓词，如果每个条件 A、B 和 C 都需要被独立地计算为 TRUE 和 FALSE，则我们需要检查 $2^3 = 8$ 种组合。如果这样的判定节点在流程图路径中出现两次，那么仅考虑这两个判定节点就有 $8 \times 8 = 64$ 种可能的组合。当然，并不是每一种组合都是可以满足的，因为某些条件之间可能有依赖关系，如当 B 为 TRUE 时，A 一定为 TRUE。

条件组合覆盖是一种比条件覆盖更为严格的覆盖准则。为满足条件组合覆盖，需要在测试用例（9.1）、（9.2）和（9.3）的基础上增加如下的测试用例，使 $x \equiv y$ 和 $z > w$ 的取值均为 FALSE。

$$\{x \mapsto 3, y \mapsto 4, z \mapsto 5, w \mapsto 6\} \tag{9.4}$$

条件组合覆盖准则的主要缺点是会引起测试用例数量爆炸。

9.2.6 路径覆盖

路径覆盖准则要求每条可执行路径都被一个测试用例覆盖到。

不幸的是，一段给定代码片段的路径数量可能是非常巨大的。因为循环可能会引起无限或者数量巨大的路径，所以我们不能要求测试包括循环的所有序列。

某些路径可能永远不能被执行到。某些判定谓词或者简单一阶公式可能永远不能计算得到指定的布尔值，某些代码片段也可能是不可达的（即死代码）。没有对应的测试用例来覆盖这些不可满足的条件。在准备测试用例或者在测试过程中，我们也会发现这个事实。总而言之，分析系统以查找死代码或不能取某些真值的谓词的问题，是不可判定的。在准备测试或者测试的过程中，识别出某部分代码不能被执行到，本身就是关于待测试程序的有价值信息。在测试过程中，可以检查实际达到的覆盖度。覆盖度可以作为测试过程质量的一个反馈，也可以用于帮助决定是否需要更进一步的测试。随后，我们将讨论如何自动地计算测试用例，该技术与程序的演绎验证中所使用的技术类似。

基于控制流的测试覆盖技术有各种限制。首先，覆盖所有的路径一般是不可行的，很难达到完全覆盖。因此，一些潜在的错误可能未被发现，它们可能隐藏在某条未能覆盖到的路径中。另一个问题是即使我们使用了一个很好的测试套件，也仅能按照流程图中的路径执行代码。在测试时检测路径覆盖情况并不总像直觉上所认为的那么简单。

使用结构化信息来覆盖程序可能会由于代码的实际编写方式（与代码应该的编写方式形成对照）引起偏差。例如，以两个执行顺序相同的指令序列为例，它们在同一判定谓词上的值也相同，或者甚至在判定谓词中每个条件的取值也都相同。在这种情况下，我们可以选取一个测试用例来代表这些执行，忽略其他的测试用例。当然，仍有可能这些执行序列相同的测试用例中只有一个能发现错误，而其他的都不能。程序员应该对这种潜在可能性区分这些情况，并且使程序在两种情况下执行不同的路径。不幸的是，所选取的覆盖准则可能导致能发现错误的执行未能被

覆盖到。

尽管可以收集到一些现存的实验统计数据，但不同覆盖准则的有效性依然是难以评估的。有人可能会提出覆盖技术应该使用概率方法（参见 9.10 节）来分析。然而，基于类似分析的增加程序可靠性的技术可能在理论上的价值更大一些。应用该类技术最大的障碍是通常不可能得到程序的合理概率分布。关于一个“普通”的程序是什么样子的，我们一般不太会有一个好的认识，比如说，在执行中某一个条件的判定值为 TRUE 的可能性是多大。

9.2.7 不同覆盖准则的比较

不同覆盖准则间有一些明显的关系。我们说一个准则包含（subsume）另一个，意思是说满足前者一定可以保证满足后者。这意味着前一个覆盖准则比后一个更为严格，因此更有可能通过增加更多的工作量来发现错误。值得注意是，由于测试用例选择的随机性，一个满足较为宽松覆盖准则的测试集可能会发现一些能满足更为严格覆盖准则的测试集所不能发现的错误。

容易发现，边覆盖是包含语句覆盖的。同样可以发现边/条件覆盖是包含边覆盖（因此也包含语句覆盖）和条件覆盖的。

显而易见，路径覆盖包含边覆盖。条件组合覆盖包含边/条件覆盖（因此也包含边/条件覆盖所包含的覆盖准则）。路径覆盖并不包含条件组合覆盖，因为我们可能不需要覆盖到判定谓词中的所有条件就能执行所有路径。相反，满足条件组合覆盖也不一定会覆盖到所有的路径（因为循环的原因）。

上面的例子可以说明条件覆盖不包含边覆盖，边覆盖也不包含条件覆盖。更令人惊奇的是，条件覆盖甚至不包含语句覆盖。这是由可能的死代码所引起的。

上述的关系如图 9.2 所示，箭头开始端的覆盖准则包含箭头指向端的覆盖准则。

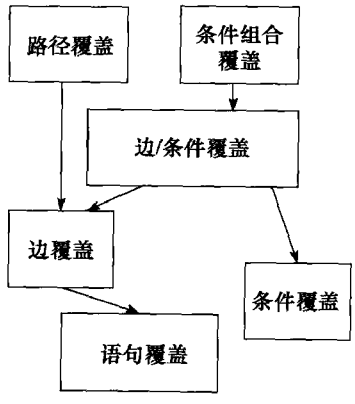


图 9.2 覆盖准则的层次关系图

9.2.8 循环覆盖

上面的覆盖准则（除了穷尽的但通常在实践中不可行的路径覆盖外）均未能充分考虑循环：它们没有提供遍历一个循环多次的规定。下面是一些用于测试循环的特定策略：

- 检查循环被跳过的情况。
- 检查循环被执行一次的情况。
- 检查循环被执行典型次数的情况（其难点在于如何评判什么次数是典型的）。
- 如果循环次数的边界值是已知的，假设为 n 次，则分别尝试执行 $n-1$、n 和 $n+1$ 次。

当待测试的循环中存在嵌套式循环时，循环的测试问题将变得更为困难。测试用例的数量将随着嵌套的层数呈指数级上升。因此，测试带嵌套的循环比测试独立循环的充分度更低。

9.3 数据流覆盖准则

数据流分析通常在对程序进行静态分析的编译器中使用。也就是说，基于其结构分析代码，而不需要先运行它。

一个使用数据流分析来检查的常见问题是某个变量在某个表达式中被（不恰当地）使用前未被初始化。这可以通过后向搜索程序的流程图来完成。当一个变量在用于计算某个表达式的语句（例如，赋值语句或判定谓词）中时，它是*活跃的*（live）。在后向遍历流程图的过程中，一个变量如果未在当前的流程图节点中被赋值，该变量就继续保持活跃。同样，如果在同一个语句

中一个变量既被使用，也被赋值，如表达式 $x:=x+1$，则该变量也保持活跃。如果变量被赋值，但没有被使用，它就死了（dead）。对一个判定节点而言，将会有两种可能性，一个变量如果在该节点的谓词表达式中被使用，或者在至少一个后继节点中被使用，它就是活跃的（注意我们的分析是从后继节点反向到判定节点进行的）。如果在达到程序的开始点时仍有一些变量是活跃的，那么这些变量可能会在未初始化前被使用。当然，由于没有运行代码，这里面可能有一些是误报；某些搜索路径可能并没有实际的执行路径与之对应。编译器仍有可能产生警告。

覆盖准则的目标是在对待测代码保持较好错误发现率的同时最小化测试用例的数量。上一节所讨论的基于控制流的覆盖准则的一个缺点是生成的测试用例与后面程序数据的处理方式并不需要一致。在 9.2 节基于控制流的覆盖准则中，可能未能包含某些执行序列，其中某个变量为了特殊目的而被赋予基个值，但随后该值被误用。针对控制流覆盖准则（除路径覆盖外），所选取的测试用例可能根本未体现出为某变量设置了一个特定的值，并在随后使用或检查该值的内容等。Rapps 和 Weyuker [121] 提出了基于变量的定义点（赋值）和使用点间路径的几种覆盖技术。

我们给出一些为表述数据流覆盖准则所需要的简单定义。对每个变量 x，在流程图（对应于程序中的语句和条件）中定义如下的几个节点集合：

$def(x)$ 是指为 x 赋值的节点（例如，在一个赋值语句或者输入语句中）。

$p\text{-}use(x)$ 是在一个谓词表达式中使用 x 的节点（例如，在一个 if 语句或 while 语句中）。

$c\text{-}use(x)$ 是指在一个除谓词表达式外其他类型表达式中使用 x 的节点（例如，在一个为某变量赋值的表达式中）。

$def\text{-}clear(x)$ 路径指的是仅包含 x 未定义的节点的路径。对每个节点 $s\in def(x)$，我们根据随后对变量 x 使用方式的差异定义如下两组节点：

$dpu(s, x)$ 包含这样的一些节点 s'，其中有一条 $def\text{-}clear(x)$ 路径从 s 到 s'（除了第一个节点，因为 $s\in def(x)$），且 s' 属于 $p\text{-}use(x)$。也就是说，有一条路径从 x 的赋值处开始，当 x 未被重新赋值时继续前进，当 x 在一个谓词表达式中被使用时结束。

$dcu(s, x)$ 包含这样的一些节点 s'，其中有一条 $def\text{-}clear(x)$ 路径从 s 到 s'，且 s' 属于 $c\text{-}use(x)$。

基于上述定义，我们可以定义不同的数据流覆盖准则。每个覆盖准则定义了在测试套件中应该包含的路径。因此，对每个程序变量 x 和对每个属于 $def(x)$ 的语句 s，除了第一个节点 s 外，需要至少包含如下的 $def\text{-}clear(x)$路径作为测试套件中的子路径：

全定义（all-defs）：包含到 $dpu(s, x)$ 或 $dcu(s, x)$ 中某节点的一条路径。

全谓词引用（all-p-uses）：到 $dpu(s, x)$ 中的每个节点均包含一条路径。

全计算引用/部分谓词引用（all-c-uses/some-p-uses）：到 $dcu(s, x)$ 中的每个节点均包含一条路径，但如果 $dcu(s, x)$ 是空的，则至少包含一条到 $dpu(s, x)$ 中某节点的路径。

全谓词引用/部分计算引用（all-p-uses/some-c-uses）：到 $dpu(s, x)$ 中的每个节点均包含一条路径，但如果 $dpu(s, x)$ 是空的，则至少包含一条到 $dcu(s, x)$ 中某节点的路径。

全引用（all uses）：包含到 $dpu(s, x)$ 和 $dcu(s, x)$ 中每个节点的一条路径。

全定义引用路径（all-du-paths）：包含到 $dpu(s, x)$ 和 $dcu(s, x)$ 中每个节点的所有路径。

除第一个和最后一个节点可能相同外，这些路径不应包含环。这很重要，因为诸如 $x:=x+1$ 的赋值语句在某表达式中定义并使用了 x；在此赋值语句中，表达式 $x+1$ 中的 c-use 代表的是前面对该变量的定义，而不是当前语句。因此，我们允许前面的定义再次出现在同样的赋值语句中。数据流覆盖准则的层次关系如图 9.3 所示。

9.4 传播路径条件

9.2节中所给出的测试用例的示例从不同的途径来覆盖图9.1中的代码片段，描述的方式是在每条路径的判定节点前为变量赋值。采用这样的方式是为了更好地说明不同覆盖准则间的差异。然而，在测试用例的中部给出变量的值显然不是一种合理的方式。对测试用例可以更为合理地描述，如通过列出路径和程序变量的初始条件。如果一条路径包含了某些用户输入，那么每个测试用例应该同样根据沿着该测试路径按照要求的输入顺序说明需要用户输入的值。如果路径是从待测试程序除入口点以外的其他位置开始的，则也需要提供开始的位置。这些值需要保证测试沿着流程图中所选择的路径执行。

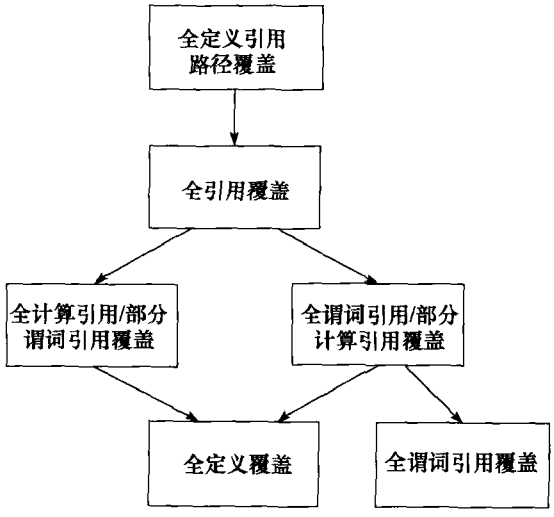


图9.3 数据流覆盖准则的层次关系

首先考虑语句覆盖准则。假定选取一组包含了待测程序所有语句的路径。这可以使用基于图的算法完成，如中国邮递员旅行算法（Chinese Postman Tour），该算法将在9.9节中进一步讨论。对每一条路径，我们需要找到一个测试用例，在路径开始处以为各个变量赋值的形式来保证执行该路径。在某些情况下，这样的变量赋值并不存在。例如，某流程图中的一条路径为变量 x 赋值2，然后判定是否满足 $x>7$，并沿着判定的 *true* 分支继续处理。静态分析可以对发现这样的路径提供帮助。此外，它们也可以在测试过程中被发现。

考虑不包含输入语句的流程图这种较为简单的情况。我们可以在给定路径的开始处计算条件，以保证执行给定路径。然后我们可以选择满足这条路径前置条件的值并用于测试该路径。下面的算法用于生成一条路径的前置条件 η。它保证当从某给定路径的开始处执行，到给定路径的结束，某给定条件 φ 应该得到满足。通过该算法计算出的条件 η 实际上是最弱前置条件。也就是说，该条件描述了所有能保证执行该给定路径并在结束时满足 φ 的程序变量的赋值。

该算法对给定的路径进行后向遍历。它从后置条件 φ 开始，并在路径中的每个点计算最弱（即最概括的）条件来完成在最后满足 φ 的路径。我们在7.1节中看到执行赋值表达式 $v:=e$ 的最弱前置条件并达到一个满足 φ 的状态是 $\varphi[e/v]$，即 φ 中 v 的每个自由出现被 e 替换。遍历判定谓词 p 的 *true* 边的最弱前置条件是 $p\wedge\varphi$。这是因为计算谓词表达式的真值不改变任何变量（除了此处不包含的程序计数器外），因此 φ 在它的执行前必须也被满足。此外，为了成功执行测试，p 必须被满足。类似地，遍历谓词判定 p 的 *false* 边的最弱前置条件是 $\neg p\wedge\varphi$。

以一个节点序列 $\xi=s_1 s_2\cdots s_n$ 为例，对路径上的每个节点 s_i，我们定义如下的符号：

type(s_i) 是 s_i 中转换的类型。类型包括开始（begin）、结束（end）、判定（decision）、赋值（assign）。

proc(s_i) 是 s_i 所属的进程。

pred(s_i) 是 s_i 的谓词表达式，在 s_i 是一个判定节点的情况下。

branch(s_i) 是 s_i 出边的标签（*true* 或 *false*），在 s_i 是一个判定节点且在该路径上有一个属于同一个进程的后继节点的情况下。否则，该节点的出边标签是未定义的（undefined）。注意如果一条路径以一个判定节点为结束，则该节点的出边不包含在该路径中。在这种情况下，该节点的判定谓词对路径条件的计算不产生影响。

$expr(s_i)$ 是赋给某变量的表达式，当 s_i 是一个赋值语句的情况下。

$var(s_i)$ 是被赋值的变量，当 s_i 是一个赋值语句的情况下。

$p[e/v]$是一个谓词 p，其中所有变量 v 的（自由）出现都被表达式 e 所替代。

```
for i := n to 1 step -1 do
  begin
    case type(s_i) do
      decision⇒
        case branch(s_i) do
          true⇒
            current_pred := current_pred ∧ pred(s_i)
          false⇒
            current_pred := current_pred ∧ ¬ pred(s_i)
          undefined⇒
            current_pred := current_pred
        end case
      assign⇒
        current_pred := current_pred [expr(s_i) /var(s_i)]
      begin, end⇒no_op
    end case;
    simplify (current_pred)
  end
```

过程 simplify 用于简化算法主循环每次迭代后所获取的公式。在简化过程中使用了一些启发式的方法，例如，结合 $x+3-2$ 中的常量，得到 $x+1$。对公式进行简化是一个难题。可以证明，在一般情况下，一个给定的整数一阶公式（或者其他常用的结构）与另一个简单公式 *true* 或 *false* 是否等价是不可判定的（参见 [95]）。前者表示该路径的执行独立于程序变量的值，后者表示路径根本不能执行。但是，该问题对某些特定的结构是可判定的。例如，Presburger 算术，即自然数（或整数）的加法、减法和比较（参见 3.6 节）。

给定一条路径，我们将 $current\_pred = true$ 作为该路径的后置条件，并使用上述算法。该算法仅找到一个前置条件，但不能找到一个满足该前置条件的赋值。

以上述摘自图 9.1 中的流程图为例，为了覆盖 y 递增语句和 x 递减语句，我们需要沿着判定节点的 *true* 边由 3 个节点所构成的路径。应用上述算法，我们从后置条件 $current\_pred = true$ 开始。所选路径的最后一个节点为 x 递减语句 $x := x-1$，即一个赋值节点。然后 $current\_pred$ 得到值 $true[x-1/x]$ 为 *true*。继续沿着该路径进行反向遍历，我们到达判定节点，该节点的谓词表达式为 $x \equiv y \wedge z > w$。因为经过了 *true* 边，我们将该谓词的值与 $current\_pred$ 中的 *true* 组合，将 $x \equiv y \wedge z > w$ 作为 $current\_pred$ 的新值。最后，我们经过 y 递增节点。因此，$current\_pred$ 变为 $x \equiv y \wedge z > w\ [y+1/y]$，即 $x \equiv y+1 \wedge z > w$。赋值

$$\{x \mapsto 2, y \mapsto 1, z \mapsto 4, w \mapsto 3\} \tag{9.5}$$

满足后一个路径前置条件。值得注意的是这个赋值对应于测试用例（9.1），那是根据判定之前而不是路径开始处的变量的值给定的。

对边覆盖的测试套件可以采用类似的方法计算，选择能覆盖每个条件的 *true* 和 *false* 边的路径。例如，上一个例子中包含赋值语句 $y := y+1$ 和具有 *false* 出边的判定节点的两个节点的路径的路径条件计算如下。我们从 $current\_pred = true$ 开始。经过判定节点的 *false* 出边，其谓词

表达式为 $x\equiv y\wedge z>w$，我们得到 $current\_pred=true\wedge\neg(x\equiv y\wedge z>w)$，可以简化为 $(x\neq y)\vee(z\leqslant w)$。经过 y 递增节点，我们得到 $current\_pred=((x\neq y)\vee(z\leqslant w))[y+1/y]$，即 $((x\neq y+1)\vee(z\leqslant w))$。对应于前述测试用例（9.2）的一个赋值是：

$$\{x\mapsto 3,y\mapsto 2,z\mapsto 5,w\mapsto 7\} \tag{9.6}$$

测试用例并不是必须从待测程序的入口处开始。它们可以通过指定程序的变量的值（包括程序计数器），在某些中间的点开始。强制测试用例总是从程序的入口开始，会使得测试更倾向于覆盖靠近入口点的语句。这可能会引入更多不必要的测试开销。另一方面，不从入口点开始测试用例存在风险，可能该测试用例的开始状态在实际执行中永远不可能被执行到。在某些特定状况下，可能会强制测试用例从属于死代码的语句开始测试。当然，从非入口点开始测试一个程序还需要增加特定的代码以提供支持。

练习 9.4.1 修改上述算法以支持条件覆盖。（提示：通过修改判定谓词来预处理所选定的路径。）

9.4.1 示例：GCD 程序

下面的 Pascal 程序接受两个自然数 $x1$ 和 $x2$，并计算它们的最大公约数（GCD），即最大的能被两个数整除的自然数。在计算的最后将结果存储在变量 $x1$ 中（同样也存储在 $x2$ 中，因为在结束时 $x1=x2$）。

```
while not x1=x2 do
    if x1>x2 then x1 := x1-x2
        else x2 := x2-x1
end
```

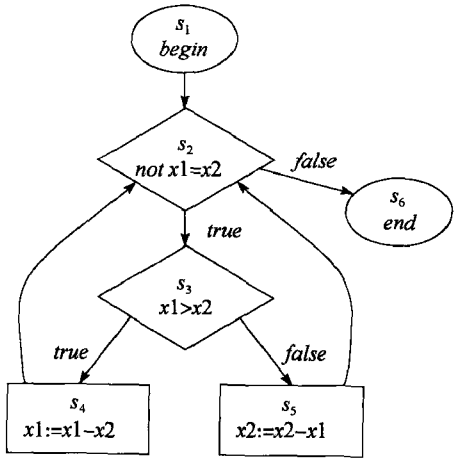


图 9.4 GCD 程序的流程图

该程序可以转换为如图 9.4 所示的流程图。我们使用路径 $\xi_1:s_1,s_2,s_3,s_4$，$\xi_2:s_1,s_2,s_3,s_5$ 和 $\xi_3:s_1,s_2,s_6$ 来覆盖该程序的所有边。

可以根据上述算法计算每条路径的路径条件。我们用一个表来展示这个计算过程。该计算过程是从每个序列的结尾以后置条件 *true* 开始反向计算的。然后在每一步，后置条件根据增加的一条语句进行相对比，计算出一个前置条件。然后对条件进行简化。执行路径 ξ_1 的过程中所计算出的条件如表 9.1 所示。

练习 9.4.2 为路径 ξ_2 和 ξ_3 提供类似的表格。为路径 $\xi_4:s_1,s_2,s_3,s_4,s_2,s_3,s_5$ 提供一个表格。

表 9.1 执行路径 ξ_1 所计算出的条件

节点	语句	后置条件	前置条件	简化后
s_4	$x1:=x1-x2$	*true*	*true*	*true*
s_3	$x1>x2$	*true*	$x1>x2$	$x1>x2$
s_2	$x1\neq x2$	$x1>x2$	$x1>x2\wedge x1\neq x2$	$x1>x2$
s_1	*begin*	$x1>x2$	$x1>x2$	$x1>x2$

练习 9.4.3 找到一组路径和该组路径的开始条件，从而保证图 7.2 中的程序满足边覆盖准则。

9.4.2 含有输入语句的路径

上述测试用例中包含赋值和判定语句，但不包含要求用户提供输入的语句。为了构造包含通过命令 *input* x 请求用户输入的测试用例，其中 x 是某个程序变量，我们需要对上述算法稍作修改 [59]。与前面的方式一样，我们后向遍历所选取的路径（根据所选择的覆盖准则）。当碰到一个变量 x 输入命令时，我们观察目前所计算出的前置条件 φ，这是从当前状态遍历到结束位置所需要满足的条件。我们需要为 x 选择一个值以满足 φ，即我们需要为 x 提供一个满足 φ 的输入值。令 $y_1 \cdots y_n$ 是 φ 中（自由）出现的所有其他变量。

然后，我们需要选择一个 x 的输入值，使 $\exists y_1 \cdots \exists y_n \varphi$ 成立。再注意一下，如何找到一个这样的值在一般情况下是不可判定的。我们选择这样的值 n。将这个值记录为输入值，当测试用例到达当前输入点时必须由测试人员提供。路径中的该输入命令被替换为赋值语句 $x := n$（因为这实际上是这个测试用例中要发生的结果）。因此，输入语句的前置条件是 $\varphi[n/x]$。现在算法像以前一样继续遍历。注意在这种方式下，输入值是反序生成的，即最后的输入最先生成。

练习 9.4.4 修改本章中计算路径条件的算法，以支持处理下面的情况 [59]：

- 并发，即允许多进程交错运行的路径，且进程间存在共享变量。注意在这种情况下，由于可能的非确定性，当控制在初始位置时，计算出的前置条件不能强制执行指定的交错路径，允许执行指定的交错路径，作为非确定性选择的其他交错路径也可能会被选择执行。
- 消息传递。

9.5 等价类划分

在规约和验证中通常将执行划分成多个无差别的执行集合。然后，从每一个集合中至少选取一个序列，而不是处理所有的执行。这通常可以简化测试和验证的过程。

在演绎和自动两种验证过程中，我们有时在从同样的偏序执行得到的交错序列之间使用等价关系。给定规约通常不区分这些交错序列。在这些情况下，我们可以利用序列之间的等价关系以提供一个更方便的证明系统，允许只对某些交错序列进行推理 [76，55，113，142]。

在测试中也使用与序列间等价性的类似原理。与构造覆盖准则的动机相同，即检查一个程序的所有执行是不切实际或者是不可能的。我们可以将执行归为等价类，同一个等价类中的序列要么都没有错误，要么都含有相同的错误。

等价类划分测试的过程在很大程度上是非形式化的。我们首先使用一个系统的输入规约或者初始条件。我们并不显式地形成执行的等价类，而是尝试猜测测试每个等价类的输入值，并将输入值收集到一起，形成测试套件。假定规约指出必须满足条件组 φ_1，...，φ_n。那么每个条件 φ_i 将执行集合划分为满足该条件和不满足该条件的两个集合（当然，我们希望后一个集合是空的）。该条件组共将执行分为 2^n 个类，在每个类中一些条件得到满足，而另一些条件不满足。

由于等价类的数目可能十分巨大，我们并不希望为所有的类都提供表示。另外，我们更愿意为一个或者少量不满足条件的类提供表示；与包含了很多违反条件的测试用例相比，那些仅包含一个违反条件的测试用例更好。这是因为用前一种方法可能很难标出同样执行中的特定错误的精确位置。我们尝试猜测出可能使单个（或少量）条件失效的测试用例的输入值。另外，我们还尝试猜测满足所有条件的测试用例。

9.6 待测代码预处理

测试人员，或者程序员自身，可能需要为测试做一些特殊的准备，包括对代码的修改。这样的预处理有助于测试过程的自动化，在测试过程中保持追踪测试结果和统计数据。预处理待测

试软件包含创建一个环境来支持测试，通常称做测试框架（test harness）。增加的代码与原有的代码一起自动运行测试用例并检查测试结果的正确性。这可能还包含为程序变量赋特定的值，并强制程序从特定的位置开始运行（不一定是初始状态）。

对代码的修改可以为覆盖的质量提供信息。当运行不同的测试时，在二级存储中分配表格来保存信息。然后这些表格被用于准备测试报告。一个表格可以包含程序中每条语句的入口之类的信息。当执行对应的语句时，增加的代码用于更新每个入口。在测试结束后，表格可以用于指导诊断是否达到了完全覆盖（在这种情况下是指语句覆盖）。如果没有，则增加新的测试用例，并重复测试过程。

可以在编译时采用与标准编译器（例如使用 UNIX cc -g 命令编译 C 程序）增加调试信息相同的方式来自动增加这样的代码。与量子物理中的海森堡原理（Heisenberg principle）一样，这样运行测试会修改待测对象的行为。新增的代码会减慢程序的运行，并改变待测程序的内存分配。但除了某些特殊的情况外，如测试时间关键的嵌入式系统，这并不是个问题。

可以在待检查代码中插入断言。新增代码负责检查这些断言，如果运行时断言失效则通知测试人员。通过这种方式，一旦这样的断言被违背，则会通知测试人员，从某个给定的状态开始执行的待测程序违反了假定的程序变量间的关系。我们可以推广上述技术，在代码中的不同位置增加多处断言，来检查程序是否违反了安全属性。新增的代码依靠测试过程中成功和失败断言的结合来判断是否有安全属性被违反。

下面是一些辅助测试过程的不同类型的专用软件包：

测试用例生成。这些工具允许为系统建模，然后使用模型并根据一定的覆盖准则来自动生成测试集。

覆盖评估。这样的工具检查指定测试套件关于给定覆盖准则所达到的覆盖度。

测试执行。这些工具用于在待测程序上执行测试用例，并报告测试结果。它们可能包含与系统的接口，如通过通信队列、因特网或者文件系统。另一种可能是测试执行也包含了对待测系统的编译，以包含如本节早些时候所述的运行测试所需的额外代码。

测试管理。这些工具维护不同的测试套件，进行版本控制，并生成测试报告。运行一个甚至多个测试时可以使用测试执行工具，但这往往是不够的。测试的结果应该被仔细地加以记录并存档。测试用例需要仔细加以维护，以便于修改系统时可重复进行回归测试。

9.7 检查测试套件

测试很少保证能发现所有甚至部分设计和实现的错误。一个度量测试套件质量的方法是使用*代码覆盖度分析*。这是根据所选取的覆盖准则针对代码比较测试套件完成的。例如，可以检查语句或者边被覆盖的百分比。比较的结果是对覆盖情况的量化度量，并指出程序的哪些部分（如语句，条件）需要增加更多的测试用例来运行。在某些情况下，覆盖度分析甚至可以找到那些为达到期望的覆盖度可能不需要的冗余的测试用例。

在本章所描述的覆盖策略中，没有一种能够保证测试是完备的。如果一个程序 P 通过了一组测试，很难估计测试过程的质量。下面这个简单的思路能够帮助评估测试套件的质量：

如果一个测试套件对两个不同的程序报告了相同的结果，那么这个测试套件很可能是不够充分的。

这是基于不同的程序很可能进行不同的计算这个事实而得出的结论。这个思路用在了*变异分析*（mutation analysis）中 [25]。给定一个待测程序 P，生成不同的变异体，即该程序的不同变化版本。变异体的出现源自审查过程中产生的问题，或者是通过结构变化（如改变标签）或者替换比较符号和逻辑运算符。

P 的测试用例被依次用于每一个变异体 P'。如果某些测试用例在 P 和 P' 上行为不同，则变异体 P' 死亡。在变异分析结束后，如果相当数量的变异体仍然活跃，则说明测试套件不恰当：没有理由相信任何一个活跃的变异体 P' 比原来的程序 P 正确性低，相反，有很好的理由相信 P 或 P' 是不正确的。当然，即使 P 通过了所有的测试，且杀死了所有的变异体，变异分析也不能保证待测程序 P 没有错误。

9.8 组合性

大型软件通常由多个团队共同开发，每个团队负责部分代码，这样可以加快开发速度。因为不同的代码片段可以独立开发，所以相同的原理可以应用于软件测试。软件测试过程通常在开发团队完成其工作前就开始了。

这种组合方式的另外一个好处是减小了待测软件的规模，以允许更好地管理软件的复杂性。这允许测试人员将注意力只集中在代码所要求的部分特性上。另一个优势是在代码的一小部分中定位错误源会更加精确。

使用这一方法的主要挑战在于程序的不同部分之间是存在交互关系的。这使得在代码的其他部分没有为测试做好准备时，仅对代码的一个部分进行测试是很困难的。假设由不同团队开发不同的多个过程，它们之间存在调用层次关系。如图 9.5 所示的调用层次关系就是这样的一个例子。其中主过程是 A。它可以调用过程 B，C 和 D。过程 B 可以调用 E，而 C 和 D 可以调用 F。

为了测试一个调用了过程 Y 的过程 X，我们可以使用 X 的测试套件，其中对 Y 的调用被视做一个原子操作。如果 Y 在测试时还不可用，我们可以编写一个简单版本的 Y，称为*测试桩*（stub），它的返回值可以用于 X 的测试。类似地，如果我们想要在 X 可用之前测试 Y，我们需要编写一个简单的代码来模拟 X 发出的对 Y 的调用。这样的代码被称为*测试驱动*（driver）。因此，在图 9.5 所示的例子中，我们可以通过编写代替 B，C 和 D 的测试桩来测试 A。我们也可以编写代替 B 的测试驱动来测试 E。当完成代码开发后，可以在前面测试时不存在的真实代码上重复进行测试。

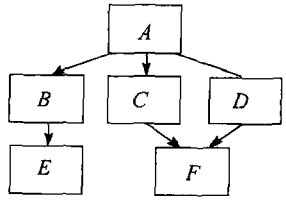


图 9.5 过程调用的层次关系

与编程一样，测试通常也是自底向上的，从层次结构的最底层开始。在我们的例子中，过程 E 和 F 首先被测试（可以由不同的团队同时对它们进行测试）。这要求为过程 B，C 和 D 编写测试驱动。一旦编写好真实的过程，它们可以使用已经编写好并测试过的过程 E 和 F，以及代替 A 的测试驱动进行测试。最后，当编写好 A 的代码时，可以使用其余的过程对其进行测试。在该例子中，为了最小化用于测试的时间，我们需要 3 个测试团队（瓶颈在于同时对 B，C 和 D 进行测试）。测试过程分为 3 个连续的阶段。最底层的节点不调用其他过程，不需要测试驱动，而其他的节点各需要一个测试驱动，一共是 4 个。

测试同样可以采用自顶向下的方式进行。在我们的例子中，这意味着为 B，C 和 D 编写测试桩，从 A 开始测试。然后，当这些过程开发好后，我们使用 E 和 F 的测试桩来测试过程 B，C 和 D。最后，使用真实代码驱动测试过程 E 和 F。这一次，我们同样需要 3 个测试团队来使测试时间最小化。除了最顶层的 A 外，每个过程各需要一个测试桩，一共是 5 个。

自底向上和自顶向下并不是仅有的两种策略，我们也可以采用非结构化的方式来测试代码。例如，通过各自增加所需的测试桩和测试驱动，同时测试过程 A 和 F。这种策略被称为*大爆炸*（big bang）。例如，为了测试 C，我们需要代表 A 的测试驱动和代表 F 的测试桩。这种非结构化的策略能够减少测试时间，因为可以同时测试更多的过程。另一方面，与结构化的策略相比较，

非结构化的策略需要编写更多的测试驱动和测试桩。在我们的例子中，需要4个测试驱动和5个测试桩。因为测试驱动和测试桩与真实的代码间可能存在偏差，所以使用更多的测试驱动和测试桩增加了在测试过程中漏报错误（或者报告不正确的错误，即误报错误）的可能性。

我们很自然地要问：应该使用何种策略来测试由多个过程、任务或者模块组成的系统。答案是测试策略应该与开发策略相适应，因为开发策略关系到测试人员可获取程序的各个部分的顺序。

9.9 黑盒测试

黑盒测试对系统进行检查时不考虑其内部结构，因此它通常局限于检查系统的功能性或特征。黑盒测试更适用于在开发阶段的后期进行的层次较高的系统测试。黑盒测试的另一个应用是用于菜单驱动的系统。测试人员可能想要检查多个菜单所提供的选项的所有可能组合是否均与规约描述的行为一致。

待建模系统预期的行为可以通过不同的方式进行描述，如文本、消息序列图（参见11.2节）等。状态机或有限自动机是将一个系统多个执行的描述封装到一起的有效手段。很多现代设计系统和方法学都是基于有限自动机（状态机）及其扩展的。该类模型适用于捕获系统的动态性质。有限自动机的研究比较多，基于它的工具也在持续增加。事实上，自动机被广泛用于设计工具（如UML）、测试工具（如TESTMASTER）和模型检验工具（COSPAN，SPIN，MURPHY）中。显然我们期望将同一个模型可应用于不同的目的，如设计、仿真、测试和验证。

黑盒测试常常基于用图或自动机对待测系统建模，并使用图算法来构造测试套件。与模型检验一样，黑盒测试的第一步是将系统建模为自动机。在模型检验中，我们自动检查模型的属性，而在这里，我们使用模型生成针对代码的测试用例。这个差别使得基于自动机的黑盒测试成为了模型检验的一个补充手段。在形成了一个关于系统的抽象模型后，两种方法可以结合。该模型被用于运行程序的测试，将系统的已知变量与它的行为进行比较。然后应用模型检验技术来自动验证它的属性[135]。此方法如图9.6所示。在图中，测试过程用于为使用模型作为基础的自动验证提供支撑。当然，测试系统是否与模型一致的事实可能不能令人完全满意。在这种情况下，需要使用开销更大但更为严格的方式（如演绎验证）来验证系统和模型间的一致性。

图9.6 一种结合测试和模型检验的方法

由于很多软件系统不局限于有限状态系统，所以建模过程可能需要进行抽象。建模过程也可以使用*扩展有限状态机*，它允许使用包含变量的条件和赋值。在这种情况下，当也考虑到变量的值时，系统可能具有无限多个状态，但只有有限多个*控制状态*。

将待测系统建模为一个自动机后，可以使用图算法来生成测试套件。与模型检验所不同的设置是测试人员在测试中不（通常是不能）进行回溯。取而代之的是，当需要在前面选择不同的选项时，测试人员需要将系统重新设定至初始状态，并将实验重新构造到需要进行不同的选择的位置。

测试生成算法使用不同的覆盖准则来生成测试套件。同样，它通常不可能覆盖到所有的路径，因为路径可能是无限的（或者数量过大）。因此，与前面所讨论的类似，使用针对有限状态系统的覆盖准则。

一个典型的基于自动机的测试生成算法尝试覆盖状态空间图中的所有边。为简便起见，我们可以假设状态空间由单个强连通分量所构成。如果不是这样，我们可以增加一些*重置*（reset）

边，以返回到初始状态。因为现在初始状态可以达到所有的其他可达状态（我们假定所有的状态都是从初始状态可达的），所以这些重置边使图成为了一个强连通分量。或者我们可以对每一个强连通分量分别进行测试，按照顺序从能到达该分量的初始状态开始。

假设我们需要覆盖图的所有边。在强连通图中，访问且仅访问每条边一次，并返回初始状态的一个充分必要条件是，所有节点的入边数与出边数相等。不是每个图都满足上述条件的。如果满足，则称该图包含一个欧拉回路（Euler cycle）。检查这个条件的方法较为简单，如果该条件满足，则可以使用一个边数的线性复杂度算法来构造这样的一条路径。

如果图不满足上述条件，我们可能要找到一条包括图中每条边至少一次的路径。因为最小化这个开销是有益的，我们可能需要找到能覆盖所有边的最少路径。这个问题称为中国邮递员问题。解决此问题是基于这样的发现：我们可以按照每条边在路径中应该出现的次数来对各条边进行复制，而不是允许一条边在路径中重复出现多次。完成这样的复制后，欧拉条件将会得到满足。然后，我们可以使用一个线性复杂度的算法［129］来在扩展后的图中找到一个欧拉回路。为满足欧拉条件，我们可以使用网络一流（network-flow）算法或者线性规划在多项式时间复杂度内找到每个节点需要复制的最小数量［39］。

覆盖每条边至少一次并不是用于构造测试套件的唯一覆盖标准。其他可能性包括：

- 覆盖所有的节点。
- 覆盖所有的路径（这通常是不现实的）。
- 覆盖 N 个节点的每个邻接序列（adjacent sequence）。
- 在测试套件的每个序列中覆盖某些节点至少一次或最多给定次数。

以如图 9.7 所示的图为例。选择两条路径就足以覆盖所有的状态，如 s_1，q_1，s_2，q_2，s_3，q_3，s_4 和 s_1，r_1，s_2，r_2，s_3，r_3，s_4。但这并不能覆盖所有的边。为了覆盖所有的边，我们需要增加路径 s_1，s_2，s_3，s_4。为了穷尽覆盖所有的路径，我们需要测试 27 条路径。这是因为节点 s_1，s_2 和 s_3 的 3 条边的选择是相互独立的。

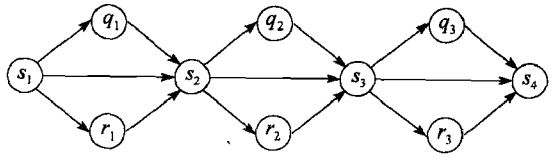


图 9.7 一个用于测试的图

值得注意的是，用于测试的覆盖准则并不能保证找到所有（或任何）错误。例如，达到满足某些条件的状态 s 后，如果在一段时间后达到满足另一些条件的状态 r，则会引发一个错误。在保证能提供访问系统所有状态（或者所有边）的测试用例的覆盖准则下，我们仍然可能无法发现错误的执行。前述的状态 s 和 r 可能会被某些测试用例所覆盖，但是这些测试用例中仅包含了两个状态中的一个（而仅一个状态自身是无法发现该错误的）。

一个部分解决方案是使用用于检查的属性，这些属性被用来为查找错误而生成测试用例。例如，我们可以采用基于自动机的模型检验思想，并使用有限自动机来表示错误的运行案例。给出待测系统的有限状态机和反例自动机，我们可以利用两者的交集生成测试用例。

9.10 概率测试

软件测试方法基于与软件验证所不同的理念：不是尝试穷尽软件的执行，而是尝试对软件的执行进行抽样。在本章前面部分所介绍的测试覆盖准则，就是为了试图获得一个合理而实用的覆盖度。为了比这个实用的方法更进一步，我们可以放弃全面测试这个目标，转而基于软件的典型使用进行测试。隐藏在这个方法背后的基本原理是：统计研究表明，所有的软件产品都是含有错误的。即使使用了不同的形式化方法技术后（一般是软件测试），仍然会有一些错误。带着这样的想法，我们应该把目标调整为在所分配的时间内使用最有效的方式来检查软件。在接受

经过测试的软件中仍然存在某些错误这个事实后，我们试图最大化“最短失效时间”（minimal time to failure，MMTF）。

为了对预期的典型使用进行测试，我们可以采用概率测试。概率测试所使用的模型是基于状态空间或有限自动机模型的一个扩展，称为马尔科夫链（Markov chain）。除了状态和转换外，它还包括从一个给定状态发生特定转换的概率。概率是介于0和1之间的实数。此外，从每个状态出发的所有转换的概率之和必须为1。

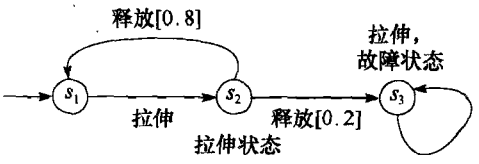


图 9.8 一个简单的马尔科夫链

如图9.8所示是一个弹簧的模型（参见图5.1和5.2节中的描述），其中增加了转换概率。因为从状态 s_1 出发仅有一个转换，从状态 s_3 出发也仅有一个转换，所以这些转换的概率为1，不需要标注。从状态 s_2 出发有两个转换，分别转换到 s_1 和 s_3。前一个转换的概率是0.8，后一个转换的概率是0.2。我们可以乘以100来通过百分比的形式表示概率。所以，每次处于状态 s_2 时，有80%的可能性转换到状态 s_1，有20%的可能性转换到 s_3。

在马尔科夫链模型中，从一个状态转换到另一个状态的概率仅依赖于当前的状态，而不依赖于截止到目前的执行历史。执行一条路径的概率就是该路径上各个转换的概率的乘积。因此，执行图9.8中路径 s_1，s_2，s_1，s_2，s_3 的概率为 $1\times0.8\times1\times0.8\times1\times0.2=0.128$，即12.8%。

值得注意的是一个马尔科夫链不过是一个模型，它不能精确地反映真实的系统。事实上，可能会有人指出上述弹簧模型的例子存在如下问题：

- 在马尔科夫链中，从一个状态转换到另一个状态的概率仅与当前状态有关。这并没有反映系统真实的属性。在弹簧的例子中，弹簧仍保持拉伸状态的概率可能与整个使用历史相关，即弹簧被拉伸了多少次。
- 一般情况下，要提供这样的概率是比较困难的。特别是当所开发的系统是全新的，没有以前关于系统使用的统计数据可用时。在弹簧的例子中，制造者可能会根据以前的研究提供各个转换的概率。然而，这样的信息通常并不存在。相反，概率通常仅反映了一个粗略的估计。

测试算法可以根据所提供的概率来对马尔科夫链进行仿真。程序设计语言通常会提供一个random()方法来生成一个仿真随机数，如生成一个介于0和1之间的数字。假定将从一个状态 s 出发的一系列转换编号为1到 n，其转换概率分别是 p_1，p_2，…，p_n。我们可以将从0到1之间的值划分为形如 $\left(\sum_{i=1}^{i=j} p_i\right) \leqslant x < \left(\sum_{i=1}^{i=j+1} p_i\right)$ 的格式（我们可以设定该区间的下边界是闭的，上边界是开的，最后一个区间的上边界也是闭的），其中 $j\in\{0,\cdots,n-1\}$，并将 p_0 设置为0。例如在弹簧的例子中，有两个区间：$0\leqslant x<0.8$ 和 $0.8\leqslant x\leqslant1$。然后，如果我们处于状态 s，random()返回的值在第 j 个区间，则我们选择 s 的第 j 个转换。

假设实现了一个无偏差的随机数生成函数，这样的仿真可以在统计意义上与待测模型的概率保持一致，并以此给出了系统所偏好的执行。

9.11 测试的优点

尽管测试并不像演绎验证或者模型检验那样是一个穷尽的方法，但它仍是最流行的用于提高软件质量的方法。主要是因为测试较为简单可行，性价比也比较高。

对软件进行验证通常需要大量的投资，比将验证作为软件开发活动中的常规活动所能承受的开销要高。模型检验已经被证明是一种验证有限状态系统的有效方法，如硬件和通信协议的验证。然而，在处理包含整型或实型变量的程序和考虑到栈或树等数据结构时，验证的方法很快

就达到了极限。在真实的系统中，一个单独的状态可能占据了计算机内存的很大一部分。模型检验的算法要求在同一时刻存放并比较多个状态，这可能是不现实的。相对而言，测试对时间和人力资源的要求通常较为合理，并可以直接在待测系统上进行测试。

9.12 测试的缺点

通常测试并没有演绎验证和模型检验那么严格。本质上，测试意味着对待测程序执行的抽样。因为测试直接应用于拥有数量巨大甚至无限多个状态的真实程序，所以使用一个系统的方法来检查程序的所有执行是不可能或者说是不切实际的。不幸的是，每一种覆盖方法都可能会漏掉一些包含错误的执行。

根据所选择的覆盖准则，白盒测试方法使用程序代码生成测试用例，这会因为系统实现的特定方式而引入偏差。以边覆盖准则为例，沿着流程图中某条路径执行，即覆盖了该路径上的边。而执行是基于检查程序中所使用的赋值语句和判定谓词来获取的。因此，实际上程序的代码和覆盖准则会为程序的执行提供一个划分建议，而测试则根据划分检查每个等价类中的一个执行。以一个覆盖流程图中某条路径的执行为例，假定待测程序有一个错误而覆盖同一条路径有两种可能性，其中一个是正确的，而另一个是不正确的。一定概率上，建议的划分所包含的执行可能是正确的，而未能包含一个针对不正确执行的测试用例。

使用有限状态机模型的黑盒测试也存在相似的问题。测试人员根据自身对系统正确和不正确行为的理解选择模型，这将会直接影响到测试套件的选择。

9.13 测试工具

可以免费用于学术目的测试工具相对较少。Brian Merick 所提供的 GCT（Generic Coverage Tool）的使用说明可从如下链接获取：

ftp://cs. uiuc. edu/pub/testng/GCT. README

注意：使用形式化方法系统通常需要填写并发送合适的发布表格，并同意某些使用条款。

9.14 扩展阅读

关于软件测试的经典书籍是：

G. J. Myers, *The Art of Software Testing*, Wiley, 1979.

其他几本 Beizer 所编写的关于测试的较全面的书籍是：

B. Beizer, *Black-Box Testing*, Wiley, 1995.

B. Beizer, *Software Testing Techniques*, 2nd edition, Van Nostrand Reinhold, 1990.

组合形式化方法

不过这只瓶上并没有写着“有毒”的字样，所以爱丽丝冒险尝了一口，觉得味道很好（它有樱桃馅饼、牛奶蛋糕、菠萝、烤火鸡、奶糖和热黄油面包混在一起的那种香味），于是很快就把它喝光了。

刘易斯·卡洛尔《爱丽丝漫游奇境记》

本书所描述的每一种软件可靠性方法都有其典型的优点与缺点。自动验证方法由于其穷尽与低人工干预的特性而备受重视。但随着被检验系统规模的增长，其效率也随之急剧下降。定理证明能被应用于无限状态系统，但它有速度慢以及需要大量人工技巧的问题。测试能够直接应用于系统，但由于存在着不完备性，因而，它可能会漏检一些错误。通过将不同的形式化方法集成起来进行协同工作，能够组合其长处，同时回避其部分短处。

10.1 抽象

对原始态的系统的验证通常太过复杂而难以实现。因此经常需要应用抽象方法，即减少系统中需要注意的细节，以获取被检验系统的简化版本。处理系统复杂度的一个成功方法是将系统简化成一个保留系统本质属性的可控版本。涉及的两个验证任务如下：

- 证明原始系统与其简化版本之间，系统的本质属性（包括我们想要验证的所有属性）得到保留。
- 证明简化版本的正确性。这个任务可在简化后通过模型检验来完成。

抽象可用于将一个无限状态系统映射为有限状态的版本，以便对其应用模型检验技术，如图 10.1 所示。此外，即使是有限状态系统也可能需要通过抽象将原始系统的状态空间减小到现有模型检验工具可控的规模。抽象通常基于应用额外的人类知识，经由人工或半自动的工具而得以实现。然而，通常情况下程序的验证是不可判定的，也无法系统地降低模型检验的复杂度，我们不能忽视这个绝对的事实。因此，我们可能无法找到合适的抽象，或是无法形式化地证明原始系统与它的抽象版本之间的对应。

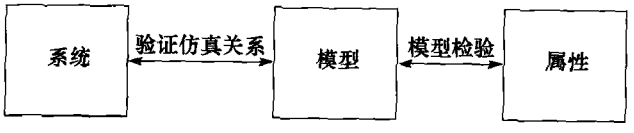


图 10.1 验证与模型检验的一种组合方法

在本节，我们将展示系统及其抽象版本之间的一些形式化关联，以及可从先抽象系统后验证抽象版本过程中得出的结论。

考虑一个实际的转换系统 P（后面会给出它的结构），其状态空间为 $\mathcal{A}^P=\langle S^P, \Delta^P, I^P, L^P, \Sigma\rangle$，这里 S^P 为它的（有限或是无限）状态集合，$\Delta^P \subseteq S^P \times S^P$ 为直接后继关系，$I^P \subseteq S^P$ 为初始状态，L 为关于状态在某字符集（状态标识）Σ 上的某标签函数。P 的抽象版本 O 对应的状态空间为 $\mathcal{A}^O=\langle S^O, \Delta^O, I^O, L^O, \Sigma\rangle$。

在第 8 章，我们看到了如何用不同的等价符号展示进程代数 agent 之间的关联性。我们同样

也能在系统的状态空间之间使用仿真的符号。抽象将以两种仿真关系呈现：前向仿真$\mathcal{R}\subseteq S^P\times S^O$与后向仿真$\mathcal{Q}\subseteq S^O\times S^P$。

前向仿真关系$\mathcal{R}$必须满足以下条件：

- 对每一个初始状态$\iota^P\in I^P$，都至少存在一个初始状态$\iota^P\in I^P$，满足$\iota^P\,\mathcal{R}\,\iota^O$。
- 若$r\,\mathcal{R}\,s$，则有$L^P(s)=L^O(r)$。
- 若$r\,\mathcal{R}\,s$，并且$(s,s')\in\Delta^P$，则至少存在一个状态$r'\in S^O$满足$(r,r')\in\Delta^O$与$r'\,\mathcal{R}\,s'$。

需要注意互相关联的状态具有同样的标签这一附加需求。对$\mathcal{R}$给定上述条件，很容易发现对$\mathcal{A}^P$的每个执行σ，$\mathcal{A}^O$总存在一个执行$\hat{\sigma}$，其标签与σ的标签匹配。也就是说，σ的第i个状态的签识$L^P(\sigma_i)$，与$L^O(\hat{\sigma}_i)$相同。因而，σ与$\hat{\sigma}$满足相同的线性时序逻辑或是Büchi自动机属性。

将状态空间$\mathcal{A}^P$与$\mathcal{A}^O$视为共用字符集Σ之上的自动机（不必有限），令$\mathcal{L}(\mathcal{A}^P)$与$\mathcal{L}(\mathcal{A}^O)$分别为$\mathcal{A}^P$与$\mathcal{A}^O$的语言，则仿真关系意味着有$\mathcal{L}(\mathcal{A}^P)\subseteq\mathcal{L}(\mathcal{A}^O)$。令$\varphi$为规约，$\mathcal{L}(\varphi)$为满足规约$\varphi$的执行集合。如果我们所验证的抽象版本$\mathcal{A}^O$满足了$\varphi$，即$\mathcal{L}(\mathcal{A}^O)\subseteq\mathcal{L}(\varphi)$，利用包含关系“$\subseteq$”的传递性，我们可以得到$\mathcal{L}(\mathcal{A}^P)\subseteq\mathcal{L}(\varphi)$。也就是说，原始系统满足被验证的属性。然而，$\mathcal{A}^O$中的某些执行可能在$\mathcal{A}^P$找不到与其匹配的执行。因此，即使在抽象版本$\mathcal{A}^O$中找到了一个不满足$\varphi$的执行反例，我们依然需要检查该执行反例是否属于$\mathcal{A}^P$。（事实上，由于$\mathcal{A}^P$本身也只是系统的一个模型，因此还需要比较该反例与所描述系统的实际执行的关联情况。）

以如图10.2所示的一个4格空间的缓冲为例。标记为*empty*的状态代表空的缓冲，标记为*full*的状态代表满的缓冲，而标记为*quasi*的状态代表处于“非空非满”状态的缓冲。在图的左边，是缓冲的状态空间$\mathcal{A}^P$，在右边则是该状态空间的抽象版本$\mathcal{A}^O$。图中从左到右的箭头对应着抽象映射关系。首先考虑如下属性：

$$\varphi=\Box((quasi\rightarrow\bigcirc(quasi\vee empty\vee full))\wedge(empty\rightarrow\bigcirc quasi)\wedge(full\rightarrow\bigcirc quasi))$$

该属性对于$\mathcal{A}^P$是满足的，并且在$\mathcal{A}^O$中得到了保持。由于仿真映射关系的存在，对$\mathcal{A}^O$——而不是$\mathcal{A}^P$——进行模型检验便能确定$\mathcal{A}^P$中φ的正确性。注意图10.2中右边的抽象版本省略了格子的数量，因此可以将它看成是对一组拥有不同格子数量的缓冲模型的抽象。现在考虑第二类属性：

$$\psi=\Box(empty\rightarrow\neg\bigcirc full\wedge\neg\bigcirc\bigcirc full\wedge\neg\bigcirc\bigcirc\bigcirc full)$$

该属性要求一个*empty*状态的缓冲不能在3步之内到达*full*状态，即缓冲应至少有4个格子。$\mathcal{A}^P$显然满足该属性，但试图证明抽象版本$\mathcal{A}^O$满足该属性则会失败。一个反例是由分别标记为*empty*、*quasi*与*full*的三个连续状态所组成的序列。然而，该反例并不在语言$\mathcal{L}(\mathcal{A}^P)$中。$\psi$就是这样一个对$\mathcal{A}^P$满足但对其抽象版本却不成立的属性例子。

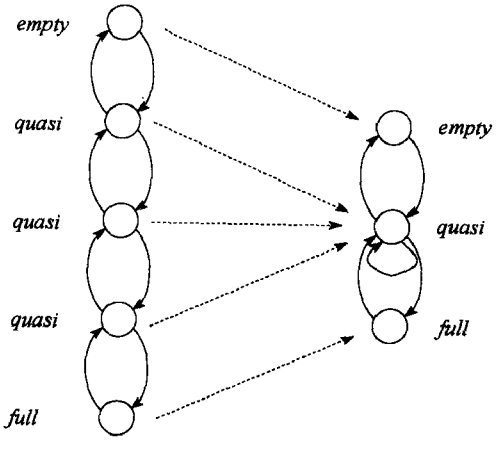


图10.2 n格缓冲及其抽象

事实上，图10.2中左边的模型忽略了存储于缓冲中的数据，因此它本身也是一种抽象。在所存数据为1比特（0或1）的简单情况下，描述该系统的一个更加详细的模型如图10.3所示。注意缓冲中最先被填入的是最左边的空格子。从使用最左边格子的元素的缓冲中移除一个元素，而后将缓冲中剩余的部分向左移动一格。在图中，由符号0、1与E（代表该格子为空）构成的序列描述缓冲中的内容。根据抽象，图10.3中的状态*EEEE*（通过仿真关系）便对应于图10.2中左边模型的*empty*状态。而那些没有空格子的状态，即图10.3中最底部的那些状态，对应于状态

full，其他状态则对应于标记为 *quasi* 的状态。

再进一步，图 10.3 中后面的模型本身是对无限状态系统的抽象。在原始系统中，缓冲的每一格可能存储着某域上的无界值，比如整数。尽管在实际实现中所存储的这些值是有界的，但取值范围可能远超出了自动验证的负载能力。再则，可为无界情况设计算法，可能需要在不考虑具体实现的实际字长的情况下进行验证。

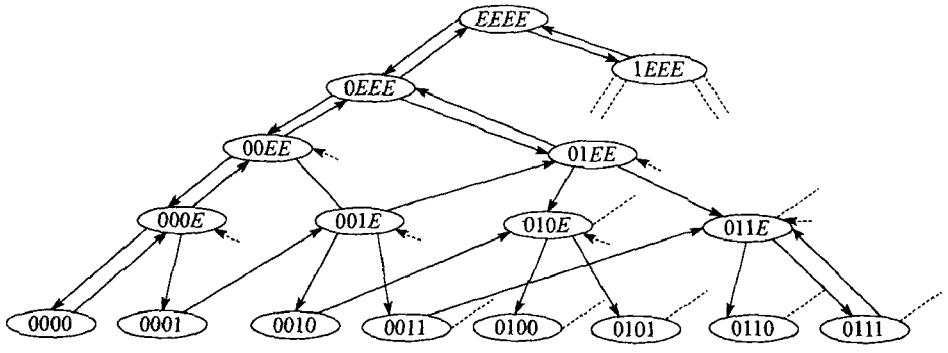


图 10.3　以比特为单位的 4 格缓冲

后向仿真关系$\mathcal{Q}$应当满足以下条件：

- 对每一个初始状态 $\iota^O \in I^O$，至少存在一个初始状态 $\iota^P \in I^P$，满足 $\iota^O \mathcal{Q} \iota^P$。
- 若 $r \mathcal{Q} S$，则有 $L^O(r) = L^P(s)$。
- 若 $r \mathcal{Q} S$，并且 $(r, r') \in \Delta^O$，则至少存在一个状态 $s' \in S^P$ 满足 $(s, s') \in \Delta^P$ 与 $r' \mathcal{Q} s'$。

给定$\mathcal{Q}$的上述条件，对$\mathcal{A}^O$ 的每个执行$\hat{\sigma}$，存在一个$\mathcal{A}^P$ 的执行 σ，其每个状态的标签都与$\hat{\sigma}$一致。则有$\mathcal{L}(\mathcal{A}^O) \subseteq \mathcal{L}(\mathcal{A}^P)$。这意味着每一个表明抽象版本$\mathcal{A}^O$ 不满足被验证属性 φ 的反例（即$\mathcal{L}(\mathcal{A}^O) \nsubseteq \mathcal{L}(\varphi)$），也是原始版本$\mathcal{A}^P$ 的反例。另一方面，即使$\mathcal{A}^O$ 所有的带标签的执行都满足规约 φ，即$\mathcal{L}(\mathcal{A}^O) \subseteq \mathcal{L}(\varphi)$，也不能得出$\mathcal{A}^P$ 同样满足 φ 的结论，因为$\mathcal{A}^P$ 中可能存在某个在$\mathcal{A}^O$ 中没有对应执行的带标签的执行。

要在状态空间（自动机）$\mathcal{A}^P$ 与$\mathcal{A}^O$ 上证明$\mathcal{R}$的仿真条件通常不现实。因此，我们试图转而在转换系统的层次上证明它。假定以 4.7 节中列出的形式给出这样两个非确定性转换系统：具体系统 $P = \langle \mathcal{S}^P, \varphi^P, \Theta^P \rangle$ 与抽象系统 $O = \langle \mathcal{S}^O, \varphi^O, \Theta^O \rangle$，分别对应于自动机$\mathcal{A}^P$ 与$\mathcal{A}^O$。

在执行转换前，将 P 的程序变量的一个有序列表表示为 $\bar{x}$，执行转换后将对应的变量有序列表表示为 $\bar{x}'$（回想一下，4.7 节中所定义的非确定性转换，它采用对应变量的两个副本来关联每个转换执行前后的值）。据此，我们以公式 $\varphi^P(\bar{x}, \bar{x}')$ 表示相应的转换。与之类似，我们用 $\bar{y}$ 与 $\bar{y}'$来分别表示执行转换前后 O 的变量有序列表，那么 $\varphi^O(\bar{y}, \bar{y}')$ 则表示 O 的转换关系。我们假定 $\bar{x}$ 与 $\bar{y}$ 是不相交的，可以通过变量重命名的方式保证这一点。我们仍然使用符号表示 $\exists \bar{x}$ 或 $\forall \bar{y}$，而不是分别列出每个变量。

令 $R(\bar{x}, \bar{y})$ 为描述抽象$\mathcal{R}$的公式，即 P 与 O 中的状态之间的关系。该公式以包含$\mathcal{G}^P$ 与$\mathcal{G}^O$的签名表达，以及以包括$\mathcal{S}^P$ 与$\mathcal{S}^O$ 的结构翻译。那么，下面两个条件是证明 P 和 O 所生成的状态空间之间满足仿真关系$\mathcal{R}$的充分条件（但并非必要条件，即可能太强）：

1. $\forall \bar{x}(\Theta^P(\bar{x}) \rightarrow \exists \bar{y}(\Theta^O(\bar{y}) \wedge R(\bar{x}, \bar{y})))$
2. $\forall \bar{x} \forall \bar{x}' \forall \bar{y}((R(\bar{x}, \bar{y}) \wedge \varphi^P(\bar{x}, \bar{x}')) \rightarrow \exists \bar{y}'(\varphi^O(\bar{y}, \bar{y}') \wedge R(\bar{x}', \bar{y}')))$

注意上述条件对 $\bar{x}$ 的每一个可能的值都进行了证实，其中可能包括了某些对应着不可达状态的值。因此，即使抽象是正确的，此条件仍然有可能不成立。假设我们可以使用 P 的一个不变量 $In(\bar{x})$，即对 P 的所有可达状态均满足的公式。这里要注意 $In(\bar{x})$ 可能允许（满足）不可

达状态。那么就有了如下较弱的条件：

1. $\forall \bar{x}((In(x) \wedge \Theta^P(\bar{x})) \rightarrow \exists \bar{y}(\Theta^O(\bar{y}) \wedge R(\bar{x},\bar{y})))$

2. $\forall \bar{x} \forall \bar{x}' \forall \bar{y}((In(x) \wedge R(\bar{x},\bar{y}) \wedge \varphi^P(\bar{x},\bar{x}')) \rightarrow \exists \bar{y}'(\varphi^O(\bar{y},\bar{y}') \wedge R(\bar{x}',\bar{y}')))$

我们尽力获得最强的可能不变量 $In(\bar{x})$。准确地满足所有可达状态（同时不满足任何不可达状态）的不变量可以令上面改写过的一对条件成为证明仿真关系的充分必要条件。然而，通常这种不变量并不总是可以找到。另一个问题是，为了能够在改写的条件中应用 $In(\bar{x})$，我们需要证明它是 P 的一个不变量。证明可以通过诸如 Manna-Pnueli 证明系统实现，特别是 7.5 节中给出的 INV 证明规则。然而，证明需要针对具体转换系统 P 完成，无法借助抽象的优势。

显而易见，为了能够推得正确性，并信赖反例，需要同时具备前向与后向仿真关系。实际上，当我们仅发现了其中一种关系时，便可以决定应用抽象过程。那么，要从模型检验的结果中得出正确的结论，就必须特别注意：如果只确定了前向仿真关系，那么就需要针对原始系统检验所得到的每个反例，以确保其不是误报。如果只使用了后向仿真关系，那么可以信赖反例，但是对 $\mathcal{A}^O$ 属性的验证并不能推得其在 $\mathcal{A}^P$ 上的正确性。

抽象通常是验证执行者基于其自身的直觉与经验以非形式化的方式完成的。这样，从抽象版本的自动验证结果中获取的结论并不安全。通过使用演绎验证工具，可以获取系统（通常是详细模型而不是实际代码）与其抽象版本之间的形式关联。我们需要在某证明系统中使用定理证明器对 $\mathcal{R}$、$\mathcal{Q}$（当其中之一不可用时则仅使用另一个）的抽象条件进行验证。还有很多种其他的方法来定义抽象。

通常没有指南说明如何选择抽象。要说明抽象保持了仿真关系（或是双向仿真关系）需要一定的技巧。有不同的“通用”抽象方法可适用于多种情况。同样，一些有用的定理也试图描述那些可通过使用多种数据类型来实现自动抽象的情况。这些*数据独立的定理* [148] 可用于自动地将一个给定的模型转换为有限状态系统。数据独立性能够被用于那些使用了某种无界数据结构，但是并不依赖其作判断的程序。一个数据独立性的例子是在某个无界域上（比如说，整数）进行有限缓冲的建模。如果被检验的属性与缓冲中所存储的元素的值无关，便可以使用图 10.2 中左侧的有限状态空间模型（或者甚至是该图中右侧更抽象的模型）将这些值抽象掉。有人建议用一些启发式方法自动获取正确的抽象，参见 [15，104，56]。下面的习题给出了一些关于这类变换的直观知识。

练习 10.1.1 思考下述针对变换系统的变换建议，并且检验其是否符合前向仿真、后向仿真或双向仿真。对这两种所建议的变换，考虑这样一个转换系统 T，它由 4.4 节中给出的确定性转换所构成。令 $\bar{z}$ 为转换系统中变量的一个子集，它满足如下条件：

> 若 x 为可能被赋予依赖 $\bar{z}$ 于中另一个变量 y 的值的任一个程序变量（即定义 T 中某个转换的表达式中有一个包含 y 的表达式被赋给 x），那么 x 同样在 $\bar{z}$ 中。举例来说，因为转换
>
> $$pc \equiv m1 \wedge x1 \geqslant 7 \rightarrow (pc,x1) := (m2,y3+4) \tag{10.1}$$
>
> 若 $y3$ 在 $\bar{z}$ 中，那么 $x1$ 必须也在该集合中。

首先，我们从转换系统中移除所有变量 $\bar{z}$，包括那些赋值给它们的表达式。则上面的转换 (10.1) 会变成

$$pc \equiv m1 \wedge x1 \geqslant 7 \rightarrow pc := m2 \tag{10.2}$$

我们同样需要改变转换的条件。我们为每个条件 c 提出了如下的变换：

1. 将 c 中每一个出现的包含 $\bar{z}$ 中变量的一阶公式替换为 *true*，转换 (10.2) 就变为

$$pc \equiv m1 \wedge true \rightarrow pc := m2$$

可简化为

$$pc \equiv m1 \rightarrow pc := m2$$

2. 将 c 中每一个出现的包含 $\bar{z}$ 中变量的一阶公式替换为 $true$，得到 $c1$。将上述每一个出现用 $false$ 替换，得到 $c2$。之后用 $c1 \vee c2$ 替换 c。

$$(pc \equiv m1 \wedge true) \vee (pc \equiv m1 \wedge false) \rightarrow pc := m2$$

同样可简化为

$$pc \equiv m1 \rightarrow pc := m2$$

（给出一个在 1 与 2 中不变换，但在简化之后成为相同转换的例子。）

3. 与上述 1 中一样，但是每次只处理一个这样的出现，直到所有的出现被替换为止。

4. 与上述 2 中一样，但是每次只处理一个这样的出现，直到所有的出现被替换为止。

10.2　组合测试与模型检验

测试技术的目的在于通过对被测系统的执行进行抽样，将其与规定的行为进行比较。已提出不同的覆盖准则以应对尽量减少测试时间与尽量增大发现错误概率之间的权衡。然而，测试并不能保证对被测系统提供穷尽覆盖。模型检验能够做到穷尽覆盖，但只能应用于被检系统的模型，而不是系统本身。由于模型与实际系统之间可能存在不一致，模型满足某些属性未必能说明实际系统也满足该属性。

本节我们将描述一项组合了模型检验的完备性，也具备像测试一样直接检测实际系统的能力的技术 [116]。将模型检验与测试的益处结合起来并不是毫无代价的，最终得到的方法具有相当高的复杂度。

10.2.1　直接检验

在某个（相当强的）假设之下，可以验证被测系统对某个规约的满足情况，例如，它与另一个系统一致，或满足某些 LTL 属性。这项技术可以被看做黑盒测试与模型检验技术的结合。所要实现的目标有两个：实现对系统的完备验证，以及在被测系统（而不是模型）上直接进行验证。这显然是期望的目标。然而，应用这类方法所依赖的假设，以及相关算法可能有着过高的复杂度，意味着模型检验与测试仍然适用于检测软件中的错误。尽管这类方法通常都涉及黑盒测试 [84]，但我们将使用*直接检验*（direct checking）这个术语来强调其与基于抽样的测试技术的不同。

直接检验的局限

模型检验与验证通常是在被验证系统的模型上进行，而不是直接应用于系统。这样做的一个原因是，如果不引入某些抽象技术的话，我们需要考虑的细节的数量会很快变得非常庞大。

但是，即使系统已经处于合理的规模之下，往往仍然会有不同的需求妨碍其直接验证。回想模型检验中所需要的不同操作：为了避免模型检验算法的无限循环，当状态空间中出现循环时，需要识别出当前所处的状态与之前出现过的状态是相同的。收集能够用于区分或识别在对系统（与理想化的模型相对）的搜索过程中遇到的两个状态的信息可能会有较高的开销，比如说，它可能需要比较被测系统转储的关键信息。在某些情况下，完整的状态信息可能甚至都不存在。例如，在硬件或是嵌入式系统中，某些状态信息在任一外部端口中可能都不直接可用。

直接检验的另一个困难之处在于对一个给定的状态在不同的环境参数下执行同样的行为有可能会走向不同的状态。这类行为非确定性可能是系统内部的计时约束或是其他非预见的外部因素导致的，例如温度或湿度。问题在于，在测试中所进行的所有实验可能都是通过一个非确定选项，但当系统被部署到域中之后，不同的环境可能会导致另一个选项成为必然选择。在这些例子中，形式化方法几乎无法给出保证。一个解决方法是，在不断地改变那些可能会造成非确定性

分支的不同的参数的情况下重复实验。但是，在缺少被测系统内部结构知识的情况下，不能完全保证所有的选项都被遍历过。

10.2.2 黑盒系统

关于黑盒测试的相关文献包括这类系统的多个变种。比如说，黑盒系统可以允许或不允许输出，或从一个给定的状态给出动作的可执行性的指示。为简单起见，我们将会研究一类特定的模型，而对其他变种的分析也照此进行。接下来，我们假定黑盒系统具有以下特性：

- 具有 n 个状态的有限状态系统。其中一个状态 ι 被确定为唯一的初始状态。或者规模就是 n，或者其规模上界是已知的。系统不会给出当前状态特征的任何指示（应用**重置**（reset）的情况除外，如下所述）。
- 动作（action）的有限字符集 Σ，规模为 d。这是用户试验该系统时可用的所有可能的输入。
- 转换关系，其中的转换用 Σ 中的动作标记。
- 存在一个动作在当前状态是否可执行的指示（可以将指示看做是 0 或 1 的输出）。如果该动作是可执行的，系统也会根据转换关系移向后继状态（后继状态可能与原状态相同）。如果不可执行的话，系统会停在同一个状态。在更一般的情况下，这可以用输出的有限集合替代。
- 可靠的**重置**能力。执行**重置**确保系统回到其唯一初始状态 ι，即重置是可靠的。我们将会在 10.2.6 节中看到为什么这个假设很重要。

实验是动作与**重置**组成的一个序列，用于验证黑盒系统的某些属性。直接检验的复杂度被定义为黑盒系统之上执行的实验的长度。有趣的是，要注意对直接检验的复杂度分析同复杂度理论中的常规分析有所不同。这是因为在一般的算法中，输入是事先已知的；而在直接检验中，则是在不完全了解黑盒系统情况的前提下，通过执行实验以改变黑盒系统的内部数据结构。因为这个缘故，对直接检验的建模经常使用不完全信息博弈论（games with incomplete information）的理论 [122]。这里将只会给出复杂度分析的非形式化描述。

10.2.3 组合锁自动机

组合锁自动机 [102，103] 在证明关于黑盒有限状态系统的实验的复杂度下界时起着重要作用。也就是说，它们提供了一个说明直接检验可能高耗时的标准例子。对每个这样的自动机，存在某个全序关系“$>$”在其状态 $s_1>s_2>\cdots>s_n$ 上，$s_1=\iota$ 为初始状态，并且状态 s_n 没有可执行的转换。对每个状态 $s_i(i<n)$，都存在一个到状态 s_{i+1} 的标记为 $\beta_i\in\Sigma$ 的转换。对所有其他的字母 $\gamma\in\Sigma\setminus\{\beta_i\}$，都存在一个从 s_i 回到初始状态的标记为 γ 的转换。这样一个自动机就是对 $\beta_1\beta_2\cdots\beta_{n-1}$ 的一个组合锁（combination lock）。图 10.4 描述了一个 $n=5$ 的组合锁自动机。

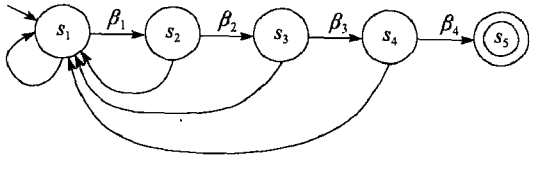


图 10.4 组合锁自动机

在组合锁自动机中，一个通向唯一无后继的状态 s_n 的序列必须有长度为 $n-1$ 的前缀，也就是 $\beta_1\beta_2\cdots\beta_{n-1}$。这是一个必要非充分条件。例如，图 10.4 中的自动机，当 $\beta_1\neq\beta_3$（或 $\beta_2\neq\beta_4$）的时候，在序列 $\beta_1\beta_2\beta_1\beta_2\beta_3\beta_4$ 之下不会达到死锁状态，因为序列中第二个 β_1 只会导致自动机回到其初始状态。

10.2.4 黑盒死锁检测

我们首先讨论一个简单的验证问题：给定一个不超过 n 个状态的确定性黑盒有限状态系统

B，我们需要检验该自动机是否会死锁，也就是到达了一个不存在任何可能的输入的状态。在这个问题中，模型中一部分是未知的，要通过实验学习。

直接而低效的解决方案是从初始状态开始系统地检验 Σ 中的所有长度为 $n-1$ 的动作序列。注意 $n-1$ 个动作有 d^{n-1} 种不同的组合方式。这使得这个问题极度困难，并且问题的下界，也就是最差情况下的复杂度不考虑同时使用少于 d^{n-1} 个动作的算法。

为了说明这一点，考虑这样一个容许在任何状态接收任何输入的黑盒自动机 B（因此也不存在死锁）。考虑一个在检查死锁过程中执行的、由 Σ 中的动作与**重置**构成的实验 ρ。假定 ρ 中出现的动作少于 d^{n-1} 个。那么至少存在一个动作序列 $\beta_1\beta_2\cdots\beta_{n-1}$ 不在关于 ρ 的实验中连续出现。如果我们以一个对 $\beta_1\beta_2\cdots\beta_{n-1}$ 的组合锁自动机 C 来取代上面的自动机 B，实验 ρ 将会针对 B 与 C 之上动作的成功与失败给出相同的指示。既然实验无法区分这两个自动机，那么我们就可能会错误地判断 C 中不存在死锁。因为随意选择了实验 ρ，就不存在长度小于 d^{n-1} 的有效选择。检验更加复杂的属性显然也不会更容易。

10.2.5　一致性测试

一致性测试（conformance testing）要将一个给定的系统模型 A 和一个黑盒系统 B 进行比较。比较成功则意味着 A 与 B 允许完全相同的序列，那么就可以通过使用模型 A 来间接检测系统 B 的属性。回想一下，黑盒自动机 B 中只有确定性转换。它会通过将动作 α 应用于当前状态来给出成功或是失败的指示。更进一步，它还允许将系统**重置**回其唯一的初始状态。（注意既然我们的模型中只有确定性动作，那么，A 和 B 的语言是等价的，这意味着根据 8.5 节中给出的等价关系，A 和 B 是等价的。）

我们初步假定 B 具有的状态数量不超过 A。所给出的算法是 B 的状态数量的多项式。我们随后将给出当只知道 B 的规模估计值的一个上界 $m>n$，并且模型 A 的状态数量依然为 n 的情况下，检验 A 和 B 之间一致性的算法。该算法与差值 $m-n$ 呈指数关系。这也是一致性测试的下界，意味着如果估计值 m 远远高于实际的状态数 n 的话，一致性测试是非常难检测的。

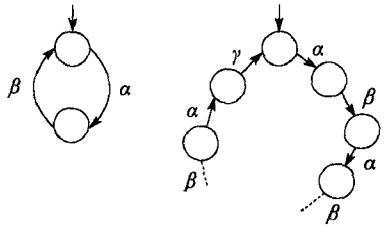


图 10.5　两个不一致的自动机

如果不知道 B 的规模的估计值，就无法通过有限的实验来保证 A 与 B 是相同的。为了说明这点，考虑图 10.5，其中左边的自动机展示了一个包含了 $\alpha\beta$ 循环（采用正则语言的语法定义的 $(\alpha\beta)^*$）的模型 A。如果测试序列的长度最大为 n 的话，那么右边的实际系统很有可能在连续重复了 $\lceil n/2 \rceil - 1$ 次 $\alpha\beta$ 后以 $\alpha\gamma$ 结束循环，而不是 $\alpha\beta$。

令 S 为具有 n 个元素的 A 的状态集合，Δ 为它的（确定性）转换关系。倘若 α 能够从 s 执行，那么我们就有 $enabled(s, \alpha)$，并且用 $\Delta(s, \alpha)$ 表示后继状态。Σ 为 A 和 B 所共用的规模为 d 的字符集。

我们假定 A 已被最小化，也就是说，不存在能够识别出与 A 相同的序列的更小的自动机。如果 A 未被最小化，那么它肯定有至少两个状态 s 与 r，不能被任意长度的实验区分开来（s 和 r 都允许完全一样的序列）。最小化确定性自动机能够通过下面的算法实现，该算法与 8.8 节中所描述的检验互仿真等价的算法相类似。（实际上，互仿真等价算法经常在模型检验中被用来实现对状态空间的最小化，用以发现那些不能被互仿真等价所区分的属性，例如分支时序逻辑 CTL，以及表达能力更强的 CTL* 属性，参见 [20, 30]。）我们在这里使用该算法不仅是为了对自动机 A 进行最小化，同时也是因为它在执行过程中生成了一组可将 A 中的状态彼此区分开来的序列集合。这些序列将会用在后面的一致性测试算法中。

对每一（无序的）对不同的状态 s 与 r，我们维护一个序列 $dist(s, r)$ 用以区分 s 与 r。也就是说，s 或 r 其中一个允许这个序列，而另一个不行。因此，$dist(s, r)=dist(r, s)$。如果 σ 是一个动作序列，α 是一个动作，那么 $\alpha.\sigma$ 表示将字符 α 加入到序列 σ 的前端。

计算区分序列

1. 初始时，对每一对 s 与 r，设置 $dist(s, r) := \varepsilon$（空序列）。
2. 对每一对不相交的状态 s，$r \in S$，以及动作 $\alpha \in \Sigma$，如满足 $enabled(s, \alpha)$ 但不满足 $enabled(r, \alpha)$，我们设置 $dist(s, r) := \alpha$。
3. 重复直至 $dist$ 条目没有发生变化为止。

   如果存在状态 s，$r \in S$ 以及动作 $\alpha \in \Sigma$ 满足

   a) $dist(s, r) := \varepsilon$

   b) $enabled(s, \alpha)$ 与 $enabled(r, \alpha)$

   c) $dist(\Delta(s, \alpha), \Delta(r, \alpha)) \neq \varepsilon$

   那么设置 $dist(s, r) := \alpha.dist(\Delta(s, \alpha), \Delta(r, \alpha))$。

因此，倘若能够从两个状态执行某些不同的动作的话，这两个状态便不能合并。同样，如果能使用序列 σ 将 s' 与 r' 区分开来，并且 s 与 r 可以通过执行动作 α 而分别到达状态 s' 与 r' 的话，那么 s 与 r 可以使用序列 $\alpha.\sigma$ 区分开来。

在算法的最后，我们可以将每个满足对任意一对状态 s，$r \in C$，均有 $dist(s, r)=\varepsilon$ 的最大状态子集 $C \subseteq S$ 合并为一个单独的状态。那么我们就能得到，如果有 $s \in C$，且 $\Delta(s, \alpha) \in C'$，C 中的每一个状态在执行了 α 之后都会转入到 C' 中的一个状态（参见 [69]）。这些所构造的状态的等价类实际上就是最小化自动机中的状态的，并且两个子集 C 与 C' 之间的转换关系也同样遵循任意两个状态 $s \in C$ 与 $r \in C'$ 之间的转换。也就是说，如果 $s \xrightarrow{\alpha} r$ 则有 $C \xrightarrow{\alpha} C'$。如果自动机 A 已经是最小化的，那么它的每一个状态等价类都是单元素类。

C 的区分集（distinguishing set）定义为：

$$ds(C) = \{dist(s,r) \mid s \in C \wedge r \notin C\}$$

最小化算法的一个属性是 C 中的每一个状态都会完全允许或不允许 $ds(C)$ 中的相同序列。进而，对其他任一个状态集合 C'，都至少在 $ds(C)$ 与 $ds(C')$ 中存在一个公共序列 σ。C' 中的所有状态执行 σ 的表现与 C 中的状态正好相反。也就是说，如果 σ 能够从 C' 的任意一个状态执行，那么它必然不能从 C 的任意一个状态执行，反之亦然。重要的是需注意每个集合 $ds(C)$ 最多能容纳 $n-1$ 条序列（因为对除了 s 本身之外的其他 $n-1$ 个状态中的每个状态至多有一条区分序列）。可见上述算法可以按如下方式分为最多 $n-1$ 个阶段执行：在每个阶段，前一个阶段的原 $dist(s, r)$ 被用于计算这个阶段的 $dist(s, r)$。每个阶段找到的新的区分序列的长度至多比前一个阶段所找到的增长一个单位。因此，$ds(C)$ 中的每条序列的长度必然小于 n。

区分集有助于构造一致性测试算法。即使 A 已经最小化，我们仍采用上述算法获取区分集。在这种情况下，每个组件 C 都包含唯一节点。因为我们要假设当 A 已被最小化，我们要用 $ds(s)$ 表示状态 s 的区分集。

一致性测试的一个任务便是检验 B 至少和 A 有同样多的状态。一个很有效的解决方法是检验是否对 A 的每个状态 r，都在 B 中存在对应的状态 s，使得 s 与 r 能够通过从它们各自自动机的初始状态执行同样的序列而达到，并且这两个状态对于 $ds(r)$ 中的每个序列的可执行性也能达成一致。为了实现对序列 $\sigma \in ds(r)$ 的检验，我们首先重置 B 为初始状态，并且根据从 A 的初始状态到状态 r 的路径在 B 上执行相应的动作。而后，我们可以在 B 上执行 σ，并检查关于从当前状态接受这个序列，B 是否与 A 有所不同。我们需要对 $ds(r)$ 中的每一个序列都重复该检验

过程。

区分集的构造规定了对 A 中的每一对状态 s 与 r，$ds(s)$ 与 $ds(r)$ 中至少存在一条公共序列，该序列仅允许从其中一个状态执行，而无法从另一个状态执行。如果 B 已经通过了上面的测试并且和 A 中的每一个区分集保持了一致，我们可以保证 B 至少有 n 个不同的状态。既然我们知道 n 同时也是 B 的规模上界，那么 B 就正好有 n 个状态。

另一个任务是检查 B 中的转换是否与 A 中的转换以相同的方式执行。假设 A 中有转换 $\Delta(r, \alpha)=r'$。我们能够对每一个序列 $\sigma \in ds(r')$ 重复下面的工作：首先，**重置** B；然后，挑选出将 A 从初始状态转换到状态 r 的动作序列并且将这个序列应用于 B；接着，我们执行动作 α，而后执行 σ 查看针对序列 σ 所得到的状态的行为是否与 r' 相同。

为了使上述所有检验系统化，我们可以为 A 构造一棵树 Tr，这棵树包括 A 的每个节点与每个转换至少一次。这样的一棵树是 A 从初始状态 ι 出发得到的生成树的一部分。因为在任何路径上，至多只有一个节点会重复两次（即最后一个节点），所以显然 Tr 的层数不超过 n。我们沿着 Tr 中的每一条路径 ρ（不需要是最大的）并且在黑盒系统 B 中进行仿真。令 r 为模型 A 在**重置**并且执行了 ρ 之后所达到的节点。我们在 B 上仿真 $ds(r)$ 中的每一个序列 σ，将其和 A 在**重置**与执行紧跟着 σ 的 ρ 之后的行为作比较。注意在检验 $ds(r)$ 的每个序列之前，我们必须重置以及执行路径 ρ 以再次到达状态 r。

Tr 的节点最多有 $2\times n\times d$ 个。对每个节点，我们需要检验最多 $n-1$ 个序列。每个这样的序列的长度最大为 $2n-1$（针对那些在树上的路径，后面紧跟着一条来自区分集的路径）。因此，测试序列的长度为 $O(d\times n^3)$。

现在考虑这样一个例子，已知 B 的状态数最大为 m，并且 $m>n$。有可能 B 并未经最小化，但依然与 A 等价。一个难以检验的黑盒系统以下面非形式化的方式描述：黑盒 B 有几个状态仿真 A 的状态，而 B 额外的 $m-n$ 个状态形成了这样一条路径 s_0，s_1，s_2，…，s_{m-n}，其行为方式类似于组合锁自动机。如果我们不是在执行组合，则转到前 n 个节点中的一个，以仿真 A 的行为。如果我们正在执行组合，则从一个节点转向下一个，直到到达最后一个节点 s_{m-n}。最后一个节点的行为与 A 中的不同，如图 10.6 所示。为了检测这样一个序列，需要通过找到组合以到达最后一个状态 s_{m-n}。

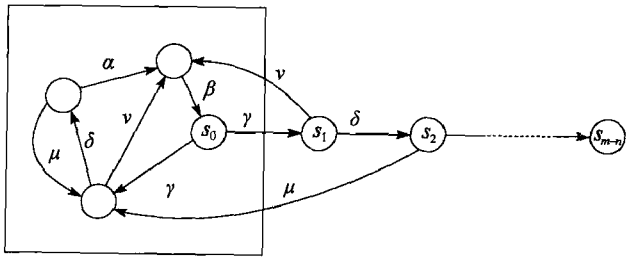


图 10.6　从黑盒自动机的一个状态出发的组合锁

本例中的算法构造了树 Tr，但试图从它的每个节点执行 $m-n$ 个或更少的动作的组合。对每个这样的组合，我们检验最后一个到达状态。如果对 A 执行**重置**，接着执行 Tr 中的一条路径，接着是一个 Σ 中 $m-n$ 个字母的序列，则令 r 为 A 中所到达的状态。从这个状态出发，我们将 $ds(r)$ 中的序列在 A 与 B 上的行为进行比较（注意对 $ds(r)$ 中的每个序列，需要重复执行**重置**和到达当前状态）。这里给出了一个长度为 $O(m^2\times n\times d^{m-n+1})$ 的实验。

10.2.6　检验重置的可靠性

在前一节，给定可靠的**重置**操作的情况下，我们给出了用于一致性测试的算法。这意味着我

们事先假定了每次都能通过使用**重置**操作将黑盒系统带回同样的初始状态。

我们现在要通过一个例子说明，即使在我们已经知道了系统 B 的内部结构这一强假设情况下，对某些系统我们还不能测试**重置**操作的可靠性。当然，如果我们对**重置**操作进行多次测试，并不能保证它之后的行为也与测试中的相同；然而，我们要说明的是，我们甚至不能提供在某些实验中**重置**行为符合预期的证据。

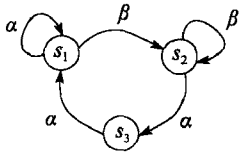


图 10.7 用于说明对**重置**操作可靠性的检验不可行的一个系统

考虑图 10.7 中的系统。假设**重置**是不可靠的，并且可能会跳向系统中 s_1，s_2，s_3 中的任一个状态。如果我们应用了**重置**，可能会转到 s_1 或 s_3，随后执行动作 α，前面的两种情况都会到达状态 s_1。所以，任何以 α 开始的测试都不能帮助我们区分这两种情况；同样，如果我们应用了**重置**，转到 s_1 或 s_2，随后执行动作 β，在两种情况下都会到达状态 s_2。所以，区分实验同样也不能以 β 开始。

那么，应用 α 或是 β 都不能确保将**重置**之后可能处于不同状态的情况相互区分开来。但是在实验中又必须从 α 或 β 中选择一个作为实验的第一个动作。如果猜得不对，该选择可能会对区分能力造成损害。Moore [102] 将这种现象作为海森堡不确定准则的离散对应。

10.2.7 黑盒检验

前几章中所讨论的模型检验与测试，是两种提高系统可靠性的相辅相成的方法。模型检验通常用于检验有限状态系统的设计是否满足某些属性（例如，互斥现象或是响应能力）。另一方面，测试通常在缺乏对内部结构的使用经验或了解的情况下应用于实际系统。它检验系统（或实现）是否符合某个设计（非形式化的说法就是，具有同样的行为）。即使可以使用被测系统的内部结构，在测试中使用内部结构信息也不是一个好主意，因为这部分信息可能会干扰测试过程。此外，整个系统可能规模庞大（例如，几百万行代码），而我们只关注系统的某些特定方面。举例来说，我们可能想要检查典型电话交换机中某个特性的具体实现，比如说“呼叫等待”，是否满足某些正确性属性。从整个系统中抽取出相关代码，尤其是对于巨大的遗留系统，几乎是不可实现的（并且其本身也容易导致错误）。

假设有意对某个系统——例如电话交换机——的特定属性进行检验。模型检验适用于检验系统模型的属性，而不是检验系统本身。另一方面，测试方法能够将系统与某抽象设计进行比较，但是通常不会用来检验特定的属性。

黑盒检验 [116] 用于处理那些验收测试需要由不知道被检验系统的设计及内部结构的用户来执行的情况。它结合了前述的两种方法，对结构未知的有限状态系统的属性进行自动检验。关于这类结构的假设与前面所讨论的黑盒一致性测试算法相同。

然而，我们另外还用规约自动机来表示被检验属性（及其否定）。它是直接给出或是如第 6 章中所示由 LTL 转换为自动机。规约自动机详细规定了允许的转换序列，相对于在建模和规约中使用至今的状态序列。黑盒检验的问题是上面所提到的模型检验问题的变体：给定了表示被检验属性不允许的执行的自动机，但没有透露被检验系统的内部结构，只有如上所述的一些实验可以在其上执行。我们仍然需要检验系统是否满足某些给定的属性。

黑盒检验工作的重心是选择合适的计算模型。与标准算法问题不同，输入并没有在计算的开始阶段给出，而是通过一组实验学习得到的。因此，常常使用不完全信息博弈理论 [122] 进行分析。这里我们不准备讨论这些问题。

在具有 d 个字母的共用字符集 Σ 之上，给定一个具有 m 个状态的规约 Büchi 自动机 A，以及一个状态数量不超过 n 的黑盒实现自动机 B，我们要检验是否存在一个序列能够同时被 A 与 B 所接受。属性自动机 A 接受坏执行，即那些不被允许的执行（参见 6.3 节）。因此，如果原先用

比如线性时序逻辑 [119] 属性 φ 给出属性，那么 A 是与 $\neg\varphi$ 对应的自动机。令 k 为 A 的状态数。注意自动机 A 不一定是确定的，并且可以有多个初始状态。

我们这里只给出一个简单的算法来执行黑盒检验 [116]。这并不是已知效率最高的算法，但却是易于理解的，并且它能够很好地展示测试与模型检验两方面技术的可能的联系。

此算法检验 Σ 上可能的动作序列。对每个序列，它从某个初始状态开始，改变已知自动机 A 的状态，同时，在对 B 进行**重置**后同样的序列被用于在黑盒自动机 B 上进行实验。我们要检查 A 与 B 的交集是否非空，并且找到该情况的反例。注意 B 中所有的状态被视为可接受，因此交集中每个状态的接受率决定于其 A 分量的接受率。我们彻底地重复这个实验，除了那些被黑盒自动机拒绝的前缀。

每个实验都由两条长度限定在 $m\times k$（因为这是交集中状态的数量）之内的路径构成。第一个序列 σ_1 需要在分量 A 接受的状态 r 处终止。第二个序列 σ_2，从 A 中的状态 r 开始执行，并且需要同样终止在状态 r 上。对每对这样的序列 σ_1、σ_2，我们执行 σ_2 多于 m 次。也就是说，我们试图执行序列 $\sigma_1(\sigma_2)^{m+1}$。如果我们在每个这样的迭代结束时成功地到达 A 中的状态 r，同时自动机 B（在**重置**后）允许执行这个序列，那么这两个自动机的交集中必定经由一个接受状态存在一个循环。这是因为接受状态 r 能够与 B 中 m 个（或是更少）状态的任一个进行配对。在这种情况下，$\sigma_1\sigma_2^{\omega}$ 构成交集的一个无限接受路径，并且可以作为生成的有限反例给出。

此算法的复杂度相当高，达到了 $O(m^2\times d^{2\times m\times k}\times k)$。这是因为这类路径有 $d^{2\times m\times k}$ 种可能的选择。每条路径的长度都限定在 $m\times k$ 内，而且我们重复了该路径 $m+1$ 次。[116] 中给出了具有更好复杂度的算法，该算法是基于在执行实验时学习自动机的结构而实现的 [9]。

10.3　净室方法

净室方法（cleanroom method）[98] 是构造可靠软件的方法，在 20 世纪 70 年代后期开始由 IBM 联合系统部的 Harlan Mills 领导的一组研究人员提出。该方法包括软件验证与测试的相关要素，并且已经被成功地应用于多个项目。该方法的主要原理与本书中先前所描述的方法没有大的差异。我们不打算给出净室方法的具体细节，只是略微描述一下其主要思想。感兴趣的读者可以查阅相关文献或本章最后列出的一本书。

10.3.1　验证

净室方法提出软件开发将在对其进行正确性验证时完成。然而，没有真正运用公理与证明规则的证明是非形式化的。净室方法的实际证明系统与 7.4 节中描述的 Hoare 逻辑在某些方面有所差异（参见 [98]），不是明确指定程序单元的前置条件与后置条件，而是给出一个公式，用来表达程序片段执行的开头与结尾处的变量之间的关系。

10.3.2　证明审查

净室方法规定审查证明，而不执行底层测试与审查代码。这是由一个小组完成的，类似于软件审查需要的（参见 9.1 节）。那么程序员的目标就是使审查组确信他的证明的正确性，这意味着代码的正确性。

10.3.3　测试

净室方法要求执行高层测试，也就是集成测试。它认为底层测试不可靠以及不够公正，并且在更可靠的验证与证明审查过程中是不必要的。实际测试是基于在 9.10 节介绍的概率测试的原理。

10.4 扩展阅读

下面的论文综述了黑盒测试的各种算法：

D. Lee, M. Yannakakis, *Principles and methods of testing finite state machines - a survey*, Proceedings of the IEEE, 84(8), 1090–1126, 1996.

下面的论文描述了一些黑盒检验算法：

D. Peled, M.Y. Vardi, M. Yannakakis, Black box checking, Formal Methods for Protocol Engineering and Distributed Systems, Formal Methods for Protocol Engineering and Distributed Systems, 1999, Kluwer, China, 225–240.

净室方法在下面的书中有所描述：

A.M. Stavely *Toward Zero-Defect Programming*, Addison-Wesley, 1999.

第11章
Software Reliability Methods

可 视 化

"嘿，没有笑的猫我倒常常看到，"爱丽丝想，"可是没有猫的笑！这是我有生以来从没见过的最奇怪的事情！"

刘易斯·卡洛尔《爱丽丝漫游奇境记》

早期的形式化方法是基于文本的，具体包括对系统的形式化表示、系统属性的规约以及与工具、测试和验证结果的交互过程。近年来，研究人员和从业人员开始意识到对软件进行可视化表示将会极大地提高软件开发效率。在许多情况下，人们通过对代码的可视化表示的观察，能够得到一些新的信息，如不同的程序对象以及它们之间的关联、控制流、通信模式等。这种直观的理解是无法通过一行一行地阅读代码获得的。

使用可视化表示法的趋势在软件开发方法学（如 UML [46]）所使用的多种图中都得到了最好的反映。这些不同种类的图可以用于描述系统的体系结构、系统中包含的不同进程或对象、进程或对象的交互、进程或对象的转换、典型和异常的执行、系统模块之间的关联等等。本章将讨论如何通过可视化表示来给形式化方法带来益处。

可视化系统所带来的好处是我们可以把呆板的语法映射成图形化对象之间的关系。例如，考虑一下，在描述自动机时，使用状态转换图或者使用包括初始及结束状态的状态转换表，哪个更易于接受？由此可见，可视化的表示在演示动态过程时有着独特的优势。再如，在表示模型检验中的某个反例时，一张标记了执行路径的流程图显然要比只是简单地列出其所执行过的代码要更加直观。

11.1 在形式化方法中运用可视化

我们已经见过了一些既可以用文本化也可以用图形化描述的符号。其一是自动机，它通过文本化或图形化的方式来描述系统的组成成分。其二是流程图，它使用图的形式展示程序。

这些示例展示了基于文字和基于图形的两种不同表示方法。这两种表示方法之间的转换都被形式化地定义过。形式化方法的工具通常允许用户与图形化的表示进行交互。对象的添加、删除、复制、大小调整、重定位、改变标签等都已经基本上成为了图形化表示方法中的标准操作。由于互联网的广泛使用，一些常见的选择或者更新操作在屏幕上都是以同样的感官展现。而这些带给用户的好处就是可以花费更少的时间来熟悉一个新的系统。

由于目前计算机对文本数据的处理仍优于对图形化数据的处理，所以基于可视化表示的工具通常需要在图形化和文本化描述形式之间做转换。基于文本的表示通常是存储在文本文件里面的。当工具需要使用这个表示的时候，需要先读入这个文本文件，然后生成图形化的表示，最终显示出来。当用户通过和工具之间的交互改变了图形化表示的时候，那么工具就要对应地修改该文本表示。

任何可视化工具都需要与操作者进行交互，因此需要响应鼠标和键盘的操作，并更新图形。一些强大的工具，例如 X-windows 系统就是为了拥有这种交互能力而开发的。近几年来，用于

简化对图形化接口进行编程的语言越来越多了，如 TCL/TCK [110]。

11.2 消息序列图

消息序列图（Message Sequence Chart，MSC）在描述通信协议的执行方面十分流行。此外，它与 UML 中描述对象间交互的用例（use-case）也密切相关 [46]。[⊖] MSC 遵循称为 ITU-Z120 [73] 的国际标准。目前，MSC 是众多用来设计通信系统 [40] 的标准描述技术之一，越来越多的工具提供了 MSC 接口 [127]。

每个消息序列图描述了一个涉及进程间通信的场景。它可以用于描述一个系统典型或异常执行的通信结构，以及在测试或模型检验中所找到的反例。这些场景描述了消息的发送与接收及其顺序。图 11.1 和图 11.2 分别是消息序列图的图形和文本表示方法。在图 11.1 中，每个进程被表示为一条竖线，竖线上方的方框内包括进程名。消息被表示为横向的箭头，从消息的发送方指向接收方。消息序列图刻画了一个消息的发送和接收事件。通常情况下，MSC 会忽略对其他事件的刻画，如判定与赋值事件。

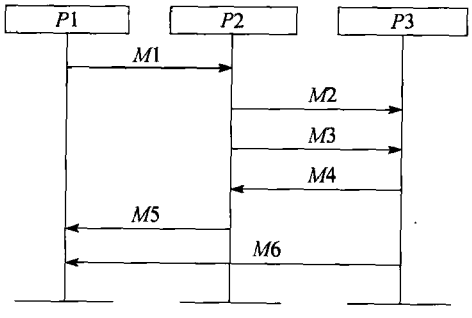


图 11.1 一个消息序列图的例子

```
msc MSC;
inst P1: process Root,
     P2: process Root,
     P3: process Root;

    instance P1;
        out M1 to P2;
        in M5 from P2;
        in M6 from P3;
    endinstance;

    instance P2;
        in M1 from P1;
        out M2 to P3;
        out M3 to P3;
        in M4 from P3;
        out M5 to P1;
    endinstance;
    instance P3;
        in M2 from P2;
        in M3 from P2;
        out M4 to P2;
        out M6 to P1;
    endinstance;
endmsc;
```

图 11.2 图 11.1 中消息序列图的文本表示方法（基于 ITU-120 标准）

支持消息序列图的一些工具，例如 Msc [8]，均支持使用图形或文本两种格式来描述消息序列图 [73]。因此消息序列图可以通过绘制图形或者输入文本的方式来生成。

我们分配给每个消息序列图一个事件的偏序集作为其语义表示。MSC 的语义取决于一些结构上的参数，这些参数在先进先出或非先进先出队列、单输入与多输入消息队列的结构中都不尽相同。在每种选择下，会在行为上体现出轻微的差别 [8]。

每个消息序列图对应一个图（S，$<$）。我们考虑它的语义解释如下：假定两个消息序列图的事件 p，$q \in S$，$p < q$ 表示 p 在 q 之前，在如下情况：

因果关系：发送事件 p 和相应的回复事件 q。

控制关系：事件 p，q 在同一进程队列中，且 p 出现在 q 之前，q 是一个消息发送事件。这种顺序反映了消息发送事件在执行前需要等待其他事件发生的情况。然而另一方面，我们一般很少对消息接收事件的顺序进行控制。

先进先出（FIFO）顺序：接收事件 p、q 在同一进程队列中，且 p 出现在接收事件 q 之前。它们的相应发送事件 p' 和 q' 也出现在同一进程队列中，且 p' 在 q' 之前。

因此，单独的消息序列图表示了事件的一个偏序集，正如在 4.13 节中讨论的。图 11.1 中的

⊖ 实际上应该是 UML 的序列图（sequence diagram）。——译者注

发送与接收事件的偏序关系如图 11.3 所示。图 11.1 中的消息序列图描述了进程 $P1$、$P2$、$P3$ 的交互过程。进程 $P1$ 向 $P2$ 发送消息 $M1$。在 $P2$ 接收到该消息后，发送两条消息 $M2$、$M3$ 到 $P3$。在 $P3$ 接收到 $M3$ 后，向 $P2$ 发送消息 $M4$，稍后再向 $P1$ 发送 $M6$。消息 $M5$ 在 $M6$ 之前被 $P1$ 接收。$M5$ 和 $M6$ 的发送事件是没有先后顺序的。

我们可以在消息序列图中应用一些简单的验证算法。其中一种是检查消息序列图中是否存在*竞争条件*（race condition）。竞争条件可以由这样的情况导致：对于包括至少一个接收事件的一组事件，我们只对其顺序进行了有限的控制（除了当两个接收者接收的消息源自同一个进程时，根据先进先出语义并不会出现问题）。举个例子，如图 11.1 所示，消息序列图中的进程 $P1$ 包含了两个消息接收事件（$M5$ 和 $M6$），由于每个进程队列（process line）是一维的（即顺序执行的），因此消息序列图的符号必须确定选择先处理其中一个接收事件，并把另一个放在后面。然而这两个消息是从 $P2$ 和 $P3$ 两个不同进程发送的，因此 $M6$ 可能比 $M5$ 先到达。所以，在使用消息序列图描述时，我们并没有理由确定接收的消息会以特定的顺序到达。

在形式上，我们将竞争定义为有这样两个消息序列图的事件 p，q：

- p 和 q 出现在同一个进程队列中；
- p 出现在 q 之前；
- 在图 $(S, \prec)$ 中不存在从 p 到 q 的路径。

在消息序列图中检测竞争情况是很简单的。我们唯一需要计算的是消息序列图中所有事件关于 $\prec^*$ 关系的传递闭包。对于事件 p，$q \in S$，仅当在图 (S, p) 中存在从 p 到 q 的路径时，有 $p \prec^* q$。然后我们只需比较在同一进程队列中的两个事件的传递闭包关系 [8]。

练习 11.2.1 在图 11.1 中进程 $P2$ 的两个接收事件是否存在竞争？请给出解释。

MSC 标准允许通过使用*高级消息序列图*（High-level Message Sequence Chart，HMSC）合并多个简单消息序列图。如图 11.4 所示，图中的每个子图都是一个单独的 MSC（或者它本身是一个 HMSC）。其中每条路径从指定的初始状态出发的，通过可见的方式连接到对应的 MSC。这就是说，当 MSC $\mathcal{S}_1$ 与 MSC $\mathcal{S}_2$ 相连接时，$\mathcal{S}_2$ 中的消息将会出现在 $\mathcal{S}_1$ 中的消息之后。有了这种功能，我们可以描述一个很大的，甚至是无限大的场景集合，每个场景可能是有限的或无限的（由于图中的循环结构）。我们可以通过使用 HMSC 来建立、调试、组织和维护 MSC 系统。

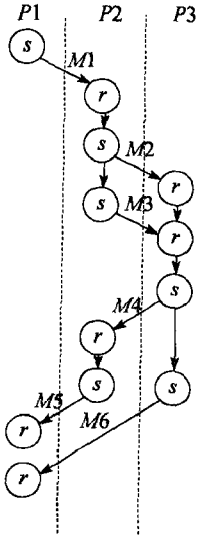


图 11.3 图 11.1 中各事件的偏序表示

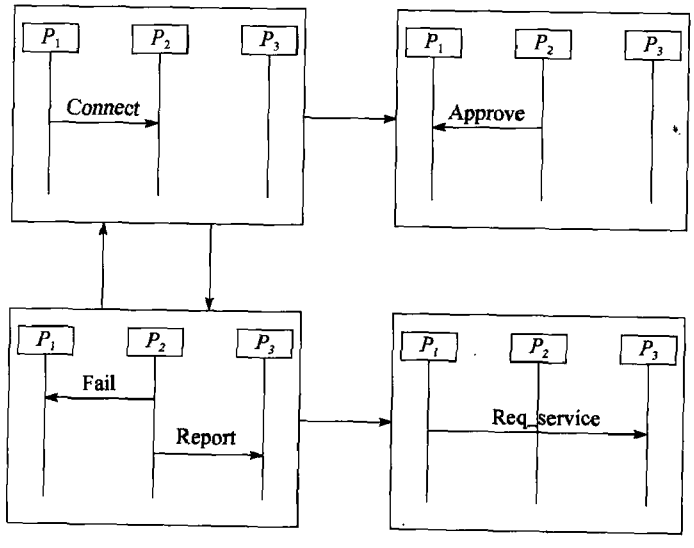


图 11.4 高级消息序列图

可以增加一种能力，以用于从 HMSC 的设计中搜索匹配给定规约的路径 [87，103]。这个规约是用相同的符号表示的，它可以是 MSC 或是 HMSC。然而，MSC（或 HMSC）的规约本身是一个模板，表示了一组事件和它们之间的顺序。一个与规约相匹配的场景至少包含了模板中的所有事件，且保留了事件之间的顺序关系。设计中匹配的路径可以比模板拥有更多的事件。（因为已证明两个 HMSC 相交的部分不可判定 [103]，所以将 HMSC 转换为模板是十分必要的。）作为搜索的结果，算法可以给出能够匹配的场景，或者给出在被检验图中找不到可匹配的场景的报告。

使用模板匹配允许我们将这类搜索机械化。可以用它来确定在设计中消息序列图是否存在不需要的属性；或者用来确定某一个需要的功能是已经被加入到现有设计中了，还是有待添加。

图 11.5 中给出了消息序列图的一个模板和匹配场景的例子。在两个图中，都有三个进程 P_1、P_2、P_3。匹配的结果是 s_2 对应 σ_1、r_2 对应 ρ_1、s_1 对应 σ_3、r_1 对应 ρ_3。

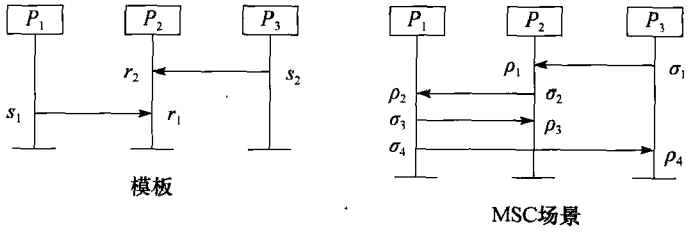


图 11.5 模板和匹配的 MSC 场景

11.3 可视化流程图和状态机

将一个程序的运行过程用自动机或流程图来表示，演示从一个状态或节点到另一个的转换过程，是一种非常有效的方式。我们可以通过高亮当前节点（改变节点的颜色或色调）来仿真执行过程。当执行一个转换时，当前节点的后继节点会替代前者成为高亮显示的节点。

当然，由于状态的数目可能会非常大或者是无限的（尽管我们可能只用它来显示状态空间中的一部分状态），因此显示整个状态空间通常是不切实际的。一种更实际的做法是用组合控制点的方式来组成表示系统的图。每个控制点对应程序计数器中一个单独的值（即使这样的值在程序的代码中是隐式表示的）。对于并发系统，我们可以在不同的窗口分开显示每个进程。在这种情况下，程序中的当前执行位置是几个进程中程序计数器的组合。因此，当仿真执行过程时，每一个活动进程中都会存在一个高亮的节点。每个转换被一个或多个进程执行。在每一个参与当前转换的进程中，当前每个节点的后继节点都会相应地替代当前节点成为被着重高亮的节点。

可视化技术可能未能提供关于当前状态的信息，而是只给出了有关当前程序计数器的信息。我们可以提供一些额外的信息，例如变量的值，或者被发送和接收的消息，通过不同的窗口显示出来。这种可视化机制可用于以下多种用途：

- 仿真系统不同的执行过程。应特别注意在仿真过程中作出的非确定性选择。这些选择可以通过使用固定的准则（如总是在列出的所有选项中选第一个）由仿真工具来确定，通过伪随机算法确定，或是委托给用户来确定。有时，工具允许用户全局性地针对整个系统或针对每个控制点来编写处理非确定性选择的程序。
- 显示模型检验或测试中找到的反例。反例是在验证或测试过程中找到的，是可以使用文本表示的一组状态和它们之间转换的序列。我们可以使用可视化方法显示它们的执行过程。

选择用于测试的执行路径。许多测试覆盖准则是基于在测试代码中选择出程序执行路径。使用可视化技术，我们可以允许用户以交互方式选择执行路径。

有许多不同的方式可以构建一个系统的可视化表示。例如我们可以使用编译器将系统的代码翻译成图 [60]。目前已有一些工具允许用户在系统建模过程中或作为系统设计的一部分生成并修改图 [12]。图中的节点或边可以包含附加信息，这些信息可以是给变量赋值的实际代码、判定谓词或发送/接收的消息。在系统设计、测试和验证的许多阶段中我们都可以使用可视化表示。它们可能会成为在开发过程中的一份有效的文档。

系统的设计可以从使用一些可视化的工具开始。有些工具可以自动生成可执行代码。虽然可能无法直接使用这些代码，但是我们可以通过修改它来开始对目标系统的开发，而不是从零开始。

图的显示是一个困难的问题，并且这本身也是一个有趣的研究领域。例如，我们希望最小化边交叉的情况。现在有一些专门程序，用于以合理的方式在图中放置不同对象，如 DOT [50]，其通常作为 UNIX 系统的一个组成部分。一些工具也可能允许用户通过鼠标，将节点从当前位置拖曳到窗口的另一个位置。在这种情况下，连接到该节点的边必须也连接到新的位置上去。近年来，我们已经见到越来越多的编程语言和开发包，如 Java 和 TCL/TK [110]，可以通过简单的拓展来支持这个功能。OBJECTTIME [127] 和 OBJECTBENCH [128] 这两个工具就是基于这种可视化表示机制的。旨在标准化和统一设计方法的 UML 方法 [46] 使用了状态空间图、消息序列图等各种图表来表示程序的执行过程。

由贝尔实验室开发的 PET 系统 [60]，可以分析使用带进程间通信的并发 Pascal 语言所编写的程序进程，并为每一个进程生成图。图中的边和节点通过调用 DOT 程序确定位置。该系统允许用户选择并发程序中进程的交错运行方式，通过使用 9.4 节中介绍过的算法计算路径条件，并显示出来。

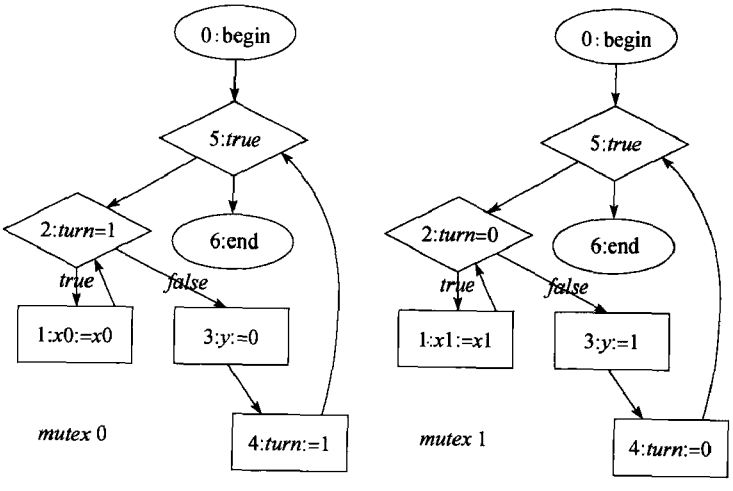


图 11.6 流程图的可视化表示

图 11.6 中的流程图描述了 *mutex*0 进程对于解决互斥访问问题的如下尝试：

```
mutex0:: while true do
  begin
    while turn=1 do
      begin
        x0 := x0
```

```
mutex1:: while true do
  begin
    while turn=0 do
      begin
        x1 := x1
```

```
    end;                                    end;
  y:=0;                                   y:=1;
  turn:=1                                 turn:=0
end                                     end
```

不允许两个进程同时访问标号为3的节点，即对 y 进行赋值的位置。通过引入一个 *turn* 变量，其值依据进程进入临界区的优先级为0或1来解决此问题。在一个进程进入临界区时，另一个进程一直等待，在标号为1、2的两个状态间转换。但是有意思的是对于下面的交错运行路径：

```
(mutex0 : 0), (mutex1 :.0), < mutex0 : 5> , < mutex1 : 5> , < mutex1 : 2> ,
< mutex0 : 2> , [mutex0 : 3], [mutex1 : 3].
```

在这种运行路径中，PET 工具返回了条件 $turn \neq 0 \wedge turn \neq 1$。这意味着当比如说 *turn* 是2的时候，两个进程同时都进入了临界区。这个错误说明了每个进程应该在不该自己进入临界区时，而不是在对方不该进入临界区时处于忙等待状态。当然，如果我们能保证 *turn* 变量是布尔类型，且不存在除此两个进程以外的其他进程，那么就可能不会出现问题。

11.4 层次状态图

对于典型的软件，其规模大、结构复杂，限制了简单状态图在表示实际系统时的实用性。简单状态图经常体现出如下不足之处：

- 状态图是*扁平化*（flat）的。因此不能自然地刻画系统的层次化结构。
- 状态图是对于被建模系统的*全局性视图*。所以，系统的并发组合需要通过各状态空间分量的乘积来表示（参见4.9节）。因此，大部分的并发结构被丢掉了。
- 由于扁平化结构和全局性的视图，其状态空间往往是巨大的。虽然状态空间爆炸问题在软件验证中已有研究，但是对于可视化来说这还是一个很大的问题：完全理解一个由几十个节点组成的图是相当困难的。
- 简单的图结构可能带来不必要的冗余。举个例子，考虑一个系统可能在任何状态下引发中断，导致中断程序执行。

使用*层次状态图*（hierarchical state graph）是解决上述问题的一个很好的办法。下面我们将展示一种特定的层次状态图符号 STATECHARTS [61]。STATECHARTS 允许将若干状态（此时应叫做*子状态*）组合成*超状态*（supers tate），如图11.7所示。同时，它还具有以下特性。

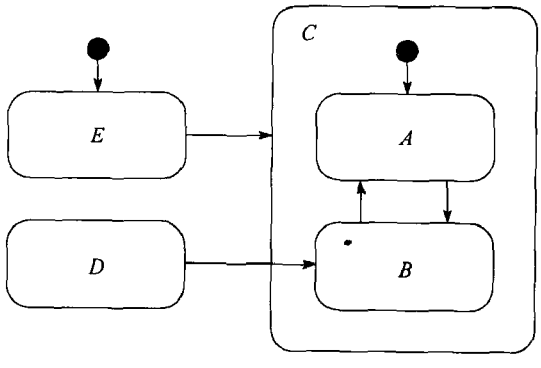


图11.7 一个简单的层次状态图

11.4.1 层次化状态

一个状态可以是一个子图。状态 s 的子图中包含的状态是 s 的*子状态*，那么 s 就是子图中状态的*超状态*。在图11.7中，状态 C 包含一个由状态 A 和 B 组成的子图。进入状态 s 的方式可以是直接由边指向其中的某一个子状态，如图11.7中 D 指向 B 的边。也可以是直接指向超状态。但在这种情况下，子图中必须存在一个默认的初始状态。其表示方法为一个实心的圆形小点上存在一条出边，并连接到初始状态。如在图11.7中，超状态 C 的默认初始状态是 A。同理可知，状态 E 的后继节点是子状态 A。从外层来看，节点 E 是初始状态。

11.4.2 统一的出口和入口

超状态之间的转换替代了它们内部的子状态间的转换。在图 11.8 中，我们可以通过将子状态 A 或 B 指向 D，而离开超状态 C。这种特性特别适用于指定中断。顶端的黑色圆点表明超状态 C 是起始超状态。另一个黑色圆点出现在 C 内部，指向子状态 A，是 C 的默认入口节点。因此，当离开 D，进入 C 时，我们会进入子状态 A。

11.4.3 并发

一个超状态可以包含若干个并发组件。当系统的超状态 s 包含并发组件时，它在每个组件中一定都处于某一个子状态。不同的并发组件用虚线划分。在图 11.9 中，超状态 S 包含了两个并发组件 C 和 F。组件 C 包含一个带有两个状态 A 和 B 的子图，同样组件 F 也包含一个带有两个状态 D 和 E 的子图。因此，当在状态 S 中时，我们必然处于 AD、AE、BD 或 BE 这些经过组合的状态中。并发组件没有必要一定用于表示实际的并发性；它也可以用于描述被分割为若干正交组件的子系统，其中每个状态是来自不同组件的本地状态的组合。

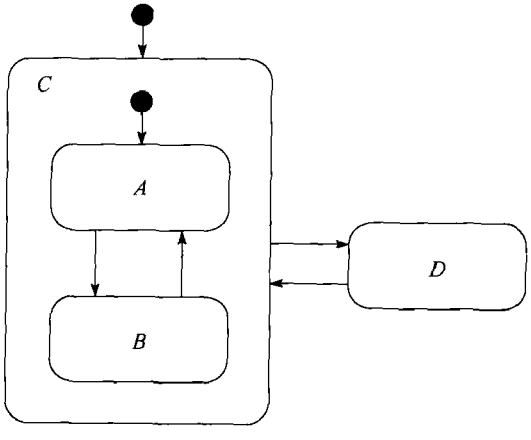


图 11.8 统一的出口

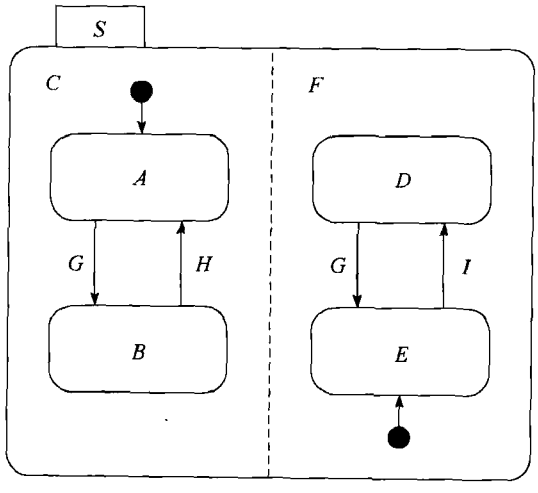


图 11.9 在一个状态中描述并发性

11.4.4 输入和输出

STATECHARTS 特别适用于描述反应式系统。因此，我们需要包含描述输入输出的符号。通过箭头来描述的每个转换，可以包含一个由系统负责接收的带变量名的输入事件，和一个由系统负责执行的输出动作。它们之间用“/”符号分开。如果没有指定输出动作，则没有必要包含“/”。在一个事件中可以包含多个输出动作，通过“;”将它们分开。另外，一个转换可以包含一个转换条件。它由“/”号起始，并包含在方括号（“[”和“]”）中。转换执行前，必须完成绑定的（可选）输入事件，并能够满足（可选）转换条件。转换执行时，绑定在转换中的（可选）动作将会被执行。

一个被标记的转换当标记它的（可选）动作出现、（可选）条件满足时，将会执行。如果某个动作标记在了不同的并发组件中的多个转换上，那么这些转换将会以同步的方式同时执行。如果某个动作只标记在一个并发组件中，那么这个动作将会独立（异步）于其他并发组件执行。因此，在图 11.9中，状态 A 到 B、状态 C 到 D 的转换是同时执行的，因为两个事件都被输入事件 G 所标记。也就是说，如果我们处于子状态 A、D，将同时转换到子状态 B、E。另一方面，

从 B 到 A 的转换被标记为 H，而从 E 到 D 的转换被标记为 I。因此，如果我们同时处于子状态 B 和 E，那么我们可以从状态 B 转换到 A，而同时保持在状态 E 中，或是从 E 转换到 D 并保持 B 状态。这表明一个组件可以在另一个组件不变时执行本组件内的状态转换。

一个条件可以关联到其他并发组件中的最新转换，即 $en(T)$ 表示其他某个组件中的最新转换是进入状态 T。类似地，$ex(T)$ 表示其他某个组件刚刚退出了状态 T。最后，$in(T)$ 表示在另一个组件中的状态是 T。

我们可以在不同组件中重用名字。为了消除名字的歧义，我们可以将子状态与超状态用点号分隔。例如，在图 11.9 中，在并发组件 S 中状态 A 是 C 的子状态，那么可以表示为 $C.A$ 或 $S.C.A$。

11.5 程序文本的可视化

虽然使用流程图来可视化地表示程序很具有吸引力，但是它也有如下一些限制：现今的程序通常规模较大，有大量的代码和上百万个状态。目前，一个很有趣的方法是使用颜色来使程序代码可视化 [13]。不同的色调表示不同等级的代码，比如用黄色表示低等级值，红色表示中等级，棕色表示高等级值等等。更深的颜色表示该代码块比其他代码块：

- 改变得更频繁；
- 被更多的测试用例覆盖；
- 在开发过程中容易出现更多的错误。

这种表示方法可以更好地用来发现哪些代码更需要应用形式化方法证明。例如，我们可能需要更重视那些经常被改变的代码，或者更希望为那些在现有测试用例中覆盖较少的代码来产生新的测试用例。有趣的是，我们可能更希望检查那些曾经被检测出存在错误的代码：因为实验表明在这些地方有更高的概率找到更多的错误。

11.6 Petri 网

Petri 网 [118] 是针对有限状态程序的一种很具吸引力的可视化表示方法。Petri 网包括通过“圆圈”表示的有限个库所（place）P，和用“细条”（bar）来表示的变迁（transition）T。边 $E \subseteq (P \times T) \cup (T \times P)$ 用来连接库所和变迁，如图 11.10 所示。Petri 网包含有限个库所的集合和有限个变迁的集合。当有一条边连接库所 p 到变迁 τ 时，p 被称做是 τ 的输入库所。当有一条边连接变迁 τ 到库所 p 时，p 被称做是 τ 的输出库所。变迁 τ 的输入库所的集合被标记为 ${}^{\cdot}\tau$，输出库所的集合被标记为 $\tau^{\cdot}$。库所可以用令牌（token）标记或者不标记。当一个变迁的所有输入库所都被标记且它的输出库所都没有被标记时，这个变迁将被允许（enabled）。变迁被允许后，变迁将被激活（执行）。这种情况下，输入库所上的令牌被消耗，同时为输出库所产生令牌。在本节中，我们将只处理基本网络系统 [140]，它在每个库所的令牌数量不多于一个。

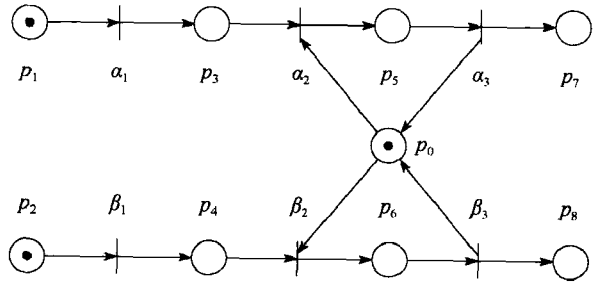


图 11.10 Petri 网

Petri 网的（全局）状态 G 是 P 中节点的一个子集。为了表明 Petri 网在某个全局状态 G 中，G 中的每个库所都用小的实心圆圈（表示令牌）标记，而非 G 中的库所则不被标记。其中一个状态被区别标记以作为初始状态。Petri 网的一次执行是从初始状态开始的最大状态序列。它通过一次次的变迁激活，获得这个状态序列里面的每个连续对。因为初始状态是独一无二的，又加上

变迁促使了序列从初始状态的改变，所以我们用变迁序列来代表 Petri 网的执行过程。

举个例子。如图 11.10 所示，有这样一个 Petri 网：它包含状态 p_0，…，p_8，和变迁：

$$T=\{\alpha_1,\alpha_2,\alpha_3,\beta_1,\beta_2,\beta_3\}$$

我们把每个状态标记为在状态中包含令牌的库所的集合。初始状态，已在图中标出，包含准确的库所位置 $G=\{p_0, p_1, p_2\}$。

结合 4.4 节中的内容，我们可以用二值变量来表示库所。一个带有令牌的库所可以表示为对应此库所的变量是 *true*，否则是 *false*。定义每个变迁 τ 为：

$$en_\tau=\bigwedge_{p\in{}^\cdot\tau} p \wedge \bigwedge_{p\in\tau^\cdot} \neg q$$

在其他变量不改变时，变迁函数 f_τ 将 ${}^\cdot\tau$ 的变量置为 *false*，将 $\tau^\cdot$ 的变量置为 *true*。

为了将 Petri 网表示为一个变迁系统，我们将使用变量 $\{p_i \mid 0\leqslant i\leqslant 8\}$。初始状态是 $\{p_0, p_1, p_2\}$。每个操作的开启条件和变迁函数如下：

$$\alpha_1: p_1 \wedge \neg p_3 \longrightarrow (p_1,p_3) := (false,true)$$

$$\alpha_2: p_0 \wedge p_3 \wedge \neg p_5 \longrightarrow (p_0,p_3,p_5) := (false,false,true)$$

$$\alpha_3: \neg p_0 \wedge p_5 \wedge \neg p_7 \longrightarrow (p_0,p_5,p_7) := (true,false,true)$$

$$\beta_1: p_2 \wedge \neg p_4 \longrightarrow (p_2,p_4) := (false,true)$$

$$\beta_2: p_0 \wedge p_4 \wedge \neg p_6 \longrightarrow (p_0,p_4,p_6) := (false,false,true)$$

$$\beta_3: \neg p_0 \wedge p_6 \wedge \neg p_8 \longrightarrow (p_0,p_6,p_8) := (true,false,true)$$

这个网络的执行过程是 $\{p_0, p_1, p_2\} \xrightarrow{\beta_1} \{p_0, p_1, p_4\} \xrightarrow{\beta_2} \{p_1, p_6\} \xrightarrow{\beta_3} \{p_0, p_1, p_8\} \xrightarrow{\alpha_1} \{p_0, p_3, p_8\} \xrightarrow{\alpha_2} \{p_5, p_8\} \xrightarrow{\alpha_3} \{p_0, p_7, p_8\}$。

在这个例子中，库所 p_0 是一个信号量 [37]。操作 α_2 和 β_2 表示获取信号量，α_3 和 β_3 表示释放信号量。

如图 11.11 所示为 4.6 节中临时互斥协议的 Petri 网模型。被标记的库所表示了系统的初始状态。使用 Petri 网的表示方式可以很方便地将协议中的属性可视化。安全属性可被定义为：无法从初始状态到达“*Critical section*1 和 *Critical section*2 都有令牌”的状态。

我们可以根据 Petri 网的激活规则，通过移动令牌来演示这个算法死锁的可能性。由于死锁将在没有可以激活的变迁时发生。那么，当被标记为 $c1:=0$ 的变迁被激活时，去掉 *Noncritical section*1 和 $c1\neq 1$ 库所的令牌，并在 *Try enter*1 和 $c1=0$ 库所加入令牌。与之对称，当被标记为 $c2:=0$ 的变迁被激活时，去掉 *Noncritical section*2 和 $c2=1$ 库所的令牌，并在 *Try enter*2 和 $c2=0$ 库所加入令牌。至此，此 Petri 网中已经不存在可以激活的变迁了。需要注意的是，因为 $c2=1$ 库所没有令牌，所以标记为 $c2=1$ 的变迁不能执行。同样，因为 $c1=1$ 库所没有令牌，所以标记为 $c1=1$ 的变迁也不能执行。

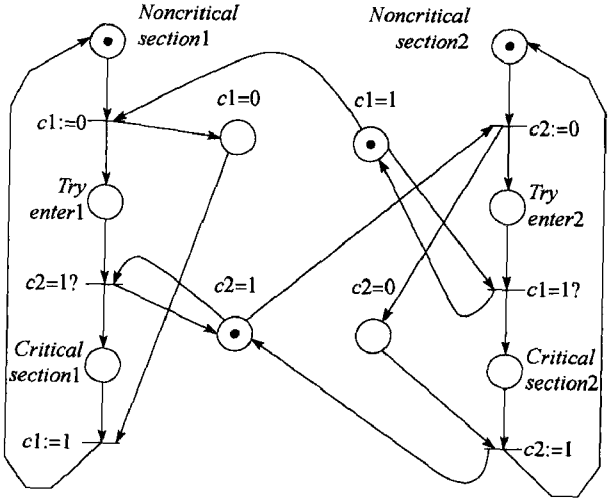


图 11.11 临时互斥算法的 Petri 网模型

练习 11.6.1 建立 4.6 节中的 Dekker 互斥算法的 Petri 网模型。

11.7 可视化工具

PEP 工具 [58]（Programming Environment based on Petri Nets，基于 Petri 网的编程环境）可以从奥尔登堡（Oldenburg）大学的以下链接获得：

http://theoretica.informatik.uni-oldenburg.de/~pep/HomePage.html

Petri 网有不同的版本，分别面向不同系统的建模与验证。一些变体包括：

- 允许一个库所包含多个令牌。
- 允许单个变迁传递多个令牌。
- 添加抑制弧（inhibitor arc），即当输入库所用抑制弧连接到一个变迁时，变迁只能在标记为空时被激活。
- 为变迁添加非量化的一阶条件。
- 用不同颜色区分令牌。

针对不同实际版本的 Petri 网模型有很多验证算法。在下一节中包含了一些综述这些算法的推荐书籍。

11.8 扩展阅读

一些 ITU 标准，如 SDL 和 MSC，可以在 ISO 文档中找到，比如：

ITU-T Recommendation Z.120，Message Sequence Chart（MSC），March 1993.

以下这本书描述了 STATECHARTS 方法：

D. Harel, M. Politi, *Modeling Reactive Systems with Statecharts*, McGraw-Hill, 1998.

Petri 网的有关书籍如下：

K. Jensen, *Coloured Petri Nets : Basic Concepts, Analysis Methods and Practical Use*, Springer-Verlag, 1995.

W. Reisig, *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*, Springer-Verlag, 1998.

W. Reisig, G. Rozenberg, eds. *Lectures on Petri Nets I: Basic Models*, Lecture Notes in Computer Science, Volume 1941, Springer-Verlag, 1998.

Booch、Jacobson 和 Rumbaugh 编制的统一建模语言（UML）是当前最流行的可视化建模框架之一。该方向的书籍有：

I. Jacobson, G. Booch, J. Rumbaugh, *Unified Software Development Process*, Addison-Wesley, 1999.

J. Rumbaugh, I. Jacobson, G. Booch, *The Unified Modeling Language, Reference Manual*, Addison-Wesley, 1998.

G. Booch, I. Jacobson, J. Rumbaugh, *The Unified Modeling Language User Guide*, Addison-Wesley, 1998.

软件系统已渐渐成为我们日常生活不可分割的一部分，我们现在每天很自然地接触这些软件系统。我们经常在万维网上购物，通过自动取款机取现金。我们几乎不再使用现金，当我们需要买东西时，这笔钱通常以数据通信方式在分布式计算机系统之间传输。计算机控制交通信号灯系统、飞机空中导航系统以及非常复杂的医疗设备。我们相信这些系统能正确工作。计算机打印输出常用做支持讨论或争议中的一些论点的证据。同时我们也期望关键系统的故障率远远低于人工操作的故障率。

现阶段，软件开发已逐步走向成熟。很明显，尽管计算机硬件不容易出错了（有时还有可能），但程序员极有可能在其代码中引入错误。可以给出关于每行代码错误率的统计数据。显然，将刚写好的代码安装到正在运行的系统中的做法不合适。这可能会导致不愉快的甚至是灾难性的后果。实际上，由错误软件造成严重破坏的事故数目在不断增加。

形式化方法用于增强系统质量，该方法尝试检测并排除软件中的错误。软件开发过程常常伴随着广泛的软件测试。测试处于形式化和非形式化方法之间。测试可以通过在软件上简单地运行任意实验来实现。它也可以使用基于数学分析的工具与技术来实现。这样，发现代码错误的几率也大大提高了。

演绎验证方法是利用数理逻辑来证明软件的详尽方法。模型检验利用算法自动验证有限状态系统，同时寻找威胁其正确性的反例。

绝对信任计算机系统是不可能的。形式化方法可以帮助我们增强对软件的信任度。发现代码中的错误是一项很难的工作。这是一个被数学家和计算机科学家所证明了的通常情况下无法彻底解决的问题。这样，不同的技术要考虑适当的折中方案也就不足为奇了。

测试技术是基于对代码执行过程进行抽样检查而实现的。即使经过非常正式的测试，代码仍可能在某些交互行为中存在一些未检测到的错误。然而测试技术是一个非常高效的方法，能够有效地发现程序中的绝大部分错误。模型检验被用来全面地检验代码是否满足某些给定的属性。这个过程基本上是自动完成的，当正确性失效时会产生一个反例。但模型检验的使用仅限于有限状态系统，因而无法对具有复杂数据结构的算法进行验证。而且，状态空间爆炸问题限制了能承受自动验证的并发组件的数量。演绎验证可以处理无限状态系统、各种各样的数据结构、有任意进程数目的系统。但整个验证过程基本上是手动的，而且速度相当慢。

形式化方法还有着其用户需要认识到的其他限制。即便完美的验证也会因人为因素而引入错误。所以，我们利用模型检验程序或者自动定理证明器来增强验证过程的可靠性。尽管如此，还是会常常在模型检验以及演绎验证工具中发现一些错误。此外，形式化方法并不直接对软件进行验证，而是对软件的抽象模型进行验证。建模过程常常是一个手动过程，易造成模型和实际系统之间的偏差。即使建模和验证都完全正确，不完备的规约仍可能会忽略软件的一些重要方面。

了解不同技术、它们的权衡和局限性后，我们便能够针对具体项目选择合适的形式化方法。单个技术往往不能满足项目的要求，最优解决方案往往是组合使用这几种技术。举例来说，对于一个银行系统，我们使用测试技术对整个系统进行测试，使用模型检验技术对其中用到的通信协议进行检验，使用演绎验证技术对处理货币交易的系统关键内核进行证明。

最近的研究主要集中在如何克服现有方法的局限性上。根据对被验证系统特性的大量观察，我们将启发式方法引入到自动验证中。在如何组合演绎和自动验证技术方面正在做大量的探索。

例如，抽象技术用来对软件建模，我们可利用演绎验证来证明抽象的正确性。随后，可以对这样的模型进行模型检验，并且使用演绎验证来支持模型检验的结果可从抽象模型推广到实际系统的结论。

随着形式化方法日趋成熟，大量软件可靠性工具逐一面世。我们现在意识到用户界面是这些工具的一个关键因素。形式化方法工具需要接受用户的表示方法，而不是给用户强加一个新的表示方法。显然，将新的形式化方法引入到软件开发生命周期中起初看上去可能会增加软件开发人员的工作量。因此，软件可靠性工具需通过易用的图形化界面吸引用户，并能够自动地完成某些复杂的任务。事实表明，图形界面和可视的形式化方法比基于文本的工具以及方法更有优势。

形式化方法利用多种形式化机制来描述系统及其属性。要根据不同情况选择适当的机制，比如简洁性、表达能力以及有效算法和工具的可用性等。选择规约形式化机制需要权衡上述因素，例如，为了更简洁或表达性更好可能会牺牲模型检验算法的效率，甚至会将自动验证过程降为手动的演绎验证过程。最近，新的形式化描述技术的标准化工作正在进行中，其中包括如SDL、LOTOS、UML等表示方法。这些表示方法都在以往的实践基础上构造，包含可视化和文本化两种描述方式。

对该研究领域而言，提高可靠性工具的效率是一个永恒的挑战。未来形式化技术可能会出现些新的、令人振奋的研究方向，例如对安全协议进行形式化验证[123]。这是受不断增长的电子商务的驱动。许多公司和金融机构都开始通过因特网提供大量的网络服务。人们可以通过家中的终端系统远程地购物、转账、玩游戏。银行账号和信用卡卡号等信息的存在、万维网的高连通性，也使得它们成为犯罪活动的目标。这些新的应用和挑战都在推动着软件可靠性技术和工具的发展。

1. M. Abadi, The power of temporal proofs, Theoretical Computer Science 65(1989), 35–83.
2. S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison-Wesley, 1995.
3. S. Aggarwal, C. Courcoubetis, P. Wolper, Adding liveness properties to coupled finite-state machines, ACM Transactions on Programming Languages and Systems 12(1990), 303–339.
4. S. Aggarwal, R. P. Kurshan, K. Sabnani, A calculus for protocol specification and validation, H. Rudin and C. H. West, (eds.), Protocol Specification, Testing and Verification, North Holland, 1983, 19–34.
5. B. Alpern, F. B. Schneider, Recognizing safety and liveness, Distributed Computing 2(1987), 117–126.
6. R. Alur, D. L. Dill: A theory of timed automata, Theoretical Computer Science, 126(1994), 183–235.
7. R. Alur, T. A. Henzinger, O. Kupferman, Alternating-time Temporal Logic, 38th Annual Symposium on Foundations of Computer Science 1997, Miami Beach, FL, 100–109.
8. R. Alur, G. J. Holzmann, D. Peled, An Analyzer for Message Sequence Charts, Tiziana Margaria, Bernhard Steffen (Eds.), Tools and Algorithms for Construction and Analysis of Systems '96, Passau, Germany, Lecture Notes in Computer Science 1055, Springer-Verlag, 1996, 35–48.
9. D. Angluin, Learning regular sets from queries and counterexamples, Information and Computation, 75(1978), 87–106.
10. K. Apt, D. Kozen, Limits for automatic verification of finite-state systems. Information Processing Letters, 15, 307–309, 1986.
11. K. R. Apt, E. R. Olderog, *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991 (second edition, 1997).
12. J. W. deBakker, *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980.
13. T. Ball, S. G. Eick, Software visualization in the large, IEEE Computer 29(4), 1996, 33–43.
14. M. Ben-Ari, *Principles of Concurrent and Distributed Programming*, Prentice-Hall, 1990.
15. S. Bensalem, Y. Lakhnech, S. Owre, Computing abstractions of infinite state systems compositionally and automatically, International Conference on Computer Aided Verification 1998, Vancouver, BC, Lecture Notes in Computer Science 1427, Springer-Verlag, 319–331.
16. G. V. Bochmann, Finite state description of communications protocols, Publication No. 236, Departement d'informatique, Universite de Montreal, July 1976.
17. T. Bolognesi, H. Brinksma, Introduction to the ISO specification language LOTOS, Computer Networks and ISDN Systems, 14(1987), 25–59.
18. H. Brinksma, H. Hermanns, Stochastic process algebra: linking process description with performances, tutorial, Proceedings of Formal Methods of Protocol Engineering and Distributed Systems '99, Beijing, China, Kluwer, 1999.
19. B. Brock, M. Kaufmann, J. Moore, ACL2 theorems about commercial microprocessors, M. Srivas and A. Camilleri (eds.), Proceedings of Formal Methods in Computer-Aided Design (FMCAD'96), Lecture Notes in Computer Science 1166, Springer-Verlag, 1996, 275–293.
20. M. C. Browne, E. M. Clarke, O. Grumberg, Characterizing finite Kripke structures in propositional temporal logic, Theoretical Computer Science, 59(1988), 115–131.
21. *Distributed Systems Analysis with CCS*, Prentice Hall, 1997.
22. J. R. Büchi. On a decision method in restricted second order arithmetic, Proceedings of the International Congress on Logic, Method and Philosophy in Science 1960, Stanford, CA, 1962. Stanford University Press, 1–12.

23. S. Budkowski, P. Dembinski, An introduction to Estelle: a specification language for distributed systems, Computer Networks and ISDN Systems, 24(1987), 3–23.
24. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. Information and Computation, 98(1992), 142–170.
25. T. A. Budd, R. J. Lipton, R. A. DeMillo, F. G. Sayward, Theoretical and empirical studies on using program mutation to test the functional correctness of programs, Proceedings of the 7th conference on Principles of Programming Languages, January 1980, 220–233.
26. C. T. Chou, D. Peled, Verifying a model-checking algorithm, Tools and Algorithms for the Construction and Analysis of Systems '96, Passau, Germany, Lecture Notes in Computer Science 1055, 1996, 241–257.
27. A. Church, A formulation of the simple theory of types, Journal of Symbolic Logic 5 (1940), 56–68.
28. E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching time temporal logic. Workshop on Logic of Programs, Yorktown Heights, NY, Lecture Notes in Computer Science 131, Springer-Verlag, 1981, 52–71.
29. E. M. Clarke, O. Grumberg, D. E. Long, Model checking and abstraction., Transactions on Programming Languages and Systems, 16(1994), 1512–1542.
30. E. M. Clarke, O. Grumberg, D. A. Peled, *Model Checking*, MIT Press, 1999.
31. R. Cleaveland, J. Parrow, B. Steffen, The Concurrency Workbench: A semantics-based tool for the verification of concurrent systems. Transactions on Programming Languages and Systems 15(1993), 36–72.
32. W. F. Clocksin, C. S. Mellish, *Programming in Prolog*, Springer-Verlag, 4th edition 1994.
33. T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.
34. C. Courcoubetis, M. Y. Vardi, P. Wolper, M. Yannakakis, Memory efficient algorithms for the verification of temporal properties, Formal Methods in System Design, Kluwer, 1(1992), 275–288.
35. J. Davies, J. C. P. Woodcock, *Using Z; Specification, Refinement and Proof*, Prentice-Hall, 1996.
36. R. Diestel, *Graph Theory*, Springer-Verlag, 2nd edition, 2000.
37. E. W. Dijkstra, Cooperating Sequential Processes, Technical Report EWD123, Technological University, Eindhoven, The Netherlands, 1965, Reprinted in: F. Genuys (Editor), Programming languages, Academic Press, London, 1968, 43–112.
38. H. D. Ebbinghaus, J. Flum, W. Thomas, *Mathematical Logic*, Undergraduate Texts in Mathematics, Springer-Verlag, Second Edition 1994.
39. J. Edmonds, E. L. Johnson, Matching, Euler tours and the Chinese postman, Mathematical Programming, 5(1973), 88–124.
40. A. Ek, J. Grabowski, D. Hogrefer, R. Jerome, B. Kosh, M. Schmitt, SDL'97, Time for testing: SDL, MSC and Trends, Proceedings of the 8th DSL Forum, Elsevier, 1997, 23–26.
41. J. Elgaard, N. Klarlund, A. Moler, MONA 1.x: New techniques for WS1S and WS2S, 10th international symposium on Computer Aided Verification, Vancouver, BC, Lecture Notes in Computer Science 1427, Springer-Verlag, 1998, 516–520
42. E. A. Emerson, E. M. Clarke, Characterizing correctness properties of parallel programs using fixpoints, International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science 85, Springer-Verlag, July 1980, 169–181.
43. E. A. Emerson, A. P. Sistla, Symmetry and model checking, Proceedings of the 5th Workshop on Computer Aided Verification, Elounda, Crete, Greece, Lecture Notes in Computer Science 697, Springer-Verlag, 1993, 463–478.
44. H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification*, Springer-Verlag, 1985.
45. R. Floyd, Assigning meaning to programs, Proceedings of symposium on applied mathematical aspects of computer science, J.T. Schwartz, ed. American Mathematical Society, 1967, 19–32.
46. M. Fowler, K. Scott, *UML Distilled : Applying the Standard Object Modeling*

Language, Addison-Wesley, 1997.
47. N. Francez, *Fairness*, Springer-Verlag, 1986.
48. N. Francez, *Program Verification*, Addison Wesley, 1992.
49. D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the Temporal Analysis of Fairness, ACM Symposium on Principles of Programming Languages, 1980, 163–173.
50. E. R. Gansner, E. Koustofios, S. C. North, K. P. Vo, A technique for drawing directed graphs, IEEE Transactions on Software Engineering 19(1993), 214–230.
51. M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979.
52. R. Gerth, D. Peled, M. Y. Vardi,, P. Wolper, Simple on-the-fly automatic verification of linear temporal logic, Protocol Specification Testing and Verification '95, 3–18, Warsaw, Chapman & Hall, 1995.
53. R. J. van Glabbeek, The linear time-branching time spectrum (extended abstract), CONCUR 1990, Theories of Concurrency, Lecture Notes in Computer Science 458, Amsterdam, Springer-Verlag, 1990, 278–297.
54. R. J. van Glabbeek, The linear time - branching time spectrum II, E. Best (ed.), 4th international conference on Concurrency theory, CONCUR 1993, Theories of Concurrency, Hildesheim, Germany, Lecture Notes in Computer Science 715, Springer-Verlag, 1993, 66–81.
55. P. Godefroid, Using partial orders to improve automatic verification methods, Proc. 2nd Workshop on Computer Aided Verification, New Brunswick, NJ, Lecture Notes in Computer Science 531, Springer-Verlag, 1990, 176–185.
56. S. Graf, H. Saidi, Construction of abstract state graphs with PVS, 9th International Conference on Computer Aided Verification, Lecture Notes in Computer Science 1254, 1997, 72–83.
57. R. L. Graham, B. L. Rothschild, J. H. Spencer, *Ramsey Theory*, Wiley, 2nd edition 1990.
58. B. Grahlmann, E. Best, PEP - more than a Petri Net tool, TACAS 1996, Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science 1055, Springer-Verlag, 397–401
59. E. L. Gunter, R. P. Kurshan, D. Peled: PET: An Interactive Software Testing Tool, Internation conference on Computer Aided Verification 2000, Chicago, IL, Lecture Notes in Computer Science 1855, Springer-Verlag, 552-556.
60. E. L. Gunter, D. Peled, Path Exploration Tool, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Amsterdam, Lecture Notes in Computer Science 1579, Springer-Verlag, 1999, 405–419.
61. D. Harel, Statecharts: a visual formalism for complex systems, Science of Computer Programming 8(1987), 231–274.
62. O. Haugen, Special Issue of Computer Networks and ISDN Systems on SDL and MSC, 28(1996).
63. C. A. R. Hoare, An axiomatic basis for computer programming, Communication of the ACM 12(1969), 576–580.
64. C. A. R. Hoare, *Communicating Sequential Processes*, Prentice-Hall.
65. G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice-Hall Software Series, 1992.
66. G. J. Holzmann, Early Fault Detection Tools, Software Concepts and Tools, 17(1996), 63–69.
67. G. J. Holzmann, D. Peled, An improvement in formal verification, Formal Description Techniques '94, Bern, Switzerland, Chapman & Hall, 1994, 197–211.
68. G. J. Holzmann, D. Peled, M. Yannakakis, On nested depth first search, Second SPIN Workshop, American Mathematical Society, 1996, 23–32.
69. J. E. Hopcroft, J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 1979.
70. W. E. Howden, *Functional Program Testing and Analysis*, McGraw-Hill, 1987.
71. G. E. Hughes, M. J. Cresswell, *A New Introduction to Modal Logic*, Routledge, 1996.
72. ISO 8807, Information Processing Systems, Open Systems Interconnection, LOTOS - A formal description technique based on temporal ordering of observational behavior.
73. ITU-T Recommendation Z.120, Message Sequence Chart (MSC), March 1993.
74. L. Jategaonkar, A. R. Meyer, Deciding true concurrency equivalences on safe, finite nets, Theoretical Computer Science 154(1966), 107–143.

75. S. Katz, Z. Manna, Logical Analysis of Programs, Communication of the ACM, 19(1976), 188–206.
76. S. Katz, D. Peled, Defining conditional independence using collapses, *Theoretical Computer Science*, 101(1992), 337–359.
77. D. Kozen, Lower bounds for natural proof systems, 18th IEEE Symposium on Foundations of Computer Science, Providence, Rhode Island, 1977, 254–266.
78. O. Kupferman, M.Y. Vardi, Verification of fair transition systems, 8th International Conference on Computer Aided Verification, New Brunswick, NJ, Lecture Notes in Computer Science 1102, Springer-Verlag, 1996, 372–382
79. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach.* Princeton University Press, 1994.
80. R. P. Kurshan, V. Levin, M. Minea, D. Peled, H. Yenigun, Verifying hardware in its software context, ICCAD'97, San Jose, CA, November 1997, 742–749.
81. R. P. Kurshan, K. L. McMillan, A structural induction theorem for processes, Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing, ACM Press, 1989, 239–247.
82. M. Z. Kwiatkowska, Fairness for Non-interleaving Concurrency, Phd. Thesis, Faculty of Science, University of Leicester, 1989.
83. L. Lamport, What good is temporal logic, R.E.A. Mason (ed.), Proceedings of IFIP Congress, North Holland, 1983, 657–668.
84. D. Lee, M. Yannakakis, Principles and methods of testing finite state machines - a survey, Proceedings of the IEEE, 84(1996), 1090–1126.
85. D. Gries, G. Levin: Assignment and Procedure Call Proof Rules. TOPLAS 2(1980), 564–579.
86. G. Levin, D. Gries, A proof technique for communicating sequential processes, Acta Informatica 15(1981), 281–302.
87. V. Levin, D. Peled. Verification of Message Sequence Charts via Template Matching, TAPSOFT (FASE)'97, Theory and Practice ofo Software Development, Lille, France, Lecture Notes in Computer Science 1214, Springer-Verlag, 1997, 652–666.
88. O. Lichtenstein, A. Pnueli, Checking that finite state concurrent programs satisfy their linear specification, 12th annual ACM symposium on Principles of Programming Languages 1985, New Orleans, Louisiana, 97–107.
89. N. A. Lynch, M. Merritt, W. Weihl, A. Fekete, *Atomic Transactions*, Morgan-Kaufmann, 1993.
90. Z. Manna, A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag, 1991.
91. Z. Manna, A. Pnueli, How to cook a temporal proof system for your pet language. 10th annual symposium on Principles of Programming Languages 1983, Austin, Texas, 141–154.
92. Z. Manna, A. Pnueli, Adequate proof principles for invariance and liveness properties of concurrent programs, Science of Computer Programming 4(1984), 257–289.
93. Z. Manna, A. Pnueli, Tools and Rules for the Practicing Verifier, In CMU Computer Science: A 25th Anniversary Commemorative, R.F. Rashid (ed.), ACM Press and Addison-Wesley, 1991, 125–159.
94. Z. Manna, R. J. Waldinger, Fundamentals of Deductive Program Synthesis. Transactions on Software Engineering 18(1992), 674–704.
95. Y. Matiyasevich, *Hilbert's 10th Problem*, MIT Press, 1993.
96. A. Mazurkiewicz, Trace theory, Advances in Petri Nets 1986, Bad Honnef, Germany, Lecture Notes in Computer Science 255, Springer-Verlag, 1987, 279–324.
97. T. F. Melham, *Introduction to Hol : A Theorem Proving Environment for Higher Order Logic*, Cambridge University Press, 1993.
98. H. D. Mills, The new math of computer programming, Communication of the ACM 18(1975), 43–48.
99. H. D. Mills, M. Dyer, R. C. Linger, Cleanroom software engineering, IEEE Software, September 1987, 19–24.
100. R. Milner, *Communication and Concurrency*, Prentice-Hall, 1995.
101. G. J. Myers, *The Art of Software Testing*, Wiley, 1979.
102. E. F. Moore, Gedanken-experiments on sequential machines, Automata Studies, Princeton University Press, 1956, 129–153.

103. A. Muscholl, D. Peled, Z. Su, Deciding properties for message sequence charts, FoSSaCS, Foundations of Software Science and Computation Structures, Lisbon, Portugal, Lecture Notes in Computer Science 1378, Springer-Verlag, 1998, 226–242.
104. K. S. Namjoshi, R. P. Kurshan, Syntactic program transformations for automatic abstractions, International Conference on Computer Aided Verification 2000, Chicago, IL, Lecture Notes in Computer Science 1855, Springer-Verlag, 2000, 435–449.
105. R. De Nicola, Extensional equivalences for transition systems, Acta Informatica 24(1987), 211–237.
106. T. Nipkow, Term rewriting and beyond – theorem proving in Isabelle, Formal Aspects of Computing, 1(1989) 320–338.
107. D. C. Oppen, A $2^{2^{2^{pn}}}$ upper bound on the complexity of Presburger arithmetic. JCSS 16(3): 323–332 (1978).
108. S. Owicki, D. Gries, An axiomatic proof technique for parallel programs, Acta Informatica 6(1976), 319–340.
109. S. Owre, S. Rajan, J. M. Rushby, N. Shankar, M. K. Srivas, PVS: Combining specification, proof checking, and model checking, International Conference on Computer Aided Verification 1996, New Brunswick, NJ, Lecture Notes in Computer Science 1102, Springer-Verlag, 1996, 411–414.
110. J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.
111. R. Paige, R. E. Tarjan: Three partition refinement algorithms. SIAM Journal of Computing 16(1987), 973–989.
112. D. Park, Concurrency and automata on infinite sequences, Theoretical Computer Science: 5th GI-Conference, Karlsruhe, Peter Deussen (ed.), Lecture Notes in Computer Science 104, Springer-Verlag, 1981, 167–183.
113. D. Peled, All from one, one for all: on model checking using representatives, Courcoubetis (ed.), 5th workshop on Computer Aided Verification, Elounda, Greece, Lecture Notes in Computer Science 697, Springer-Verlag, 1993, 409–423.
114. D. Peled, S. Katz, A. Pnueli, Specifying and proving serializability in temporal logic, International Symposium on Logic in Computer Science, Amsterdam, IEEE Computer Society Press, 1991, 232–244.
115. D. Peled, A. Pnueli, Proving partial order properties, Theoretical Computer Science, 126(1994), 143–182.
116. D. Peled, M.Y. Vardi, M. Yannakakis, Black box checking, Formal Methods for Protocol Engineering and Distributed Systems, Formal Methods for Protocol Engineering and Distributed Systems, 1999, Kluwer, China, 225–240.
117. D. Peled, L. Zuck, From model checking to a temporal proof, The 8th International SPIN Workshop on Model Checking of Software (SPIN'2001), Toronto, Canada, Lecture Notes in Computer Science 2057, Springer Verlag.
118. C. A. Petri, Kommunikation mit Automaten, Bonn, Institut für Instrumentelle Matematik, Schriften des IIM Nr. 2(1962).
119. A. Pnueli, The temporal logic of programs, 18th IEEE symposium on Foundation of Computer Science, 1977, 46–57.
120. J. P. Quielle, J. Sifakis, Specification and verification of concurrent systems in CESAR, Proceedings of the 5th International Symposium on Programming, 1981, 337–350.
121. S Rapps, E. J. Weyuker, Selecting software test data using data flow information, IEEE Transactions on software engineering, SE-11 4(1985), 367–375.
122. J. H. Reif, The complexity of two-player games of incomplete information, Journal of Computer and System Sciences, 29(1984), 274–301.
123. P. Ryan, S. Schneider, *Modelling and Analysis of Security Protocols*, Addison-Wesley, 2001.
124. S. Safra, On the complexity of omega-automata, Proceedings of the 29th IEEE Symposium on Foundations of Computer Science, White Plains, October 1988, 319–327.
125. W. J. Savitch, Relationships between nondeterministic and deterministic tape complexities. Journal of Computer and System Sciences 4(1970), 177–192.
126. F. B. Schneider, *On Concurrent Programming*, Springer-Verlag, 1997.
127. B. Selic, G. Gullekson, P. T. Ward, J. McGee, *Real-time object-oriented modeling*, Wiley, 1994.

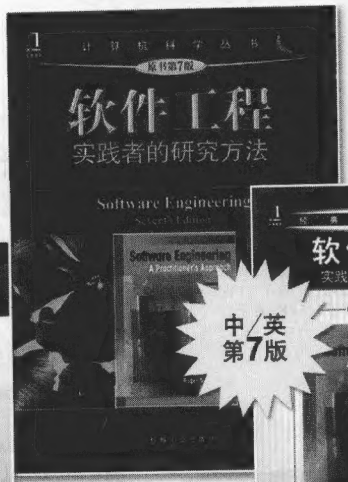
128. SES Inc., *Objectbench Technical Reference*, Scientific and Engineering Software, 1997.
129. J. P. M. Silva, Search algorithms for satisfiability problems in combinatorial switching circuits, Phd. Dissertation, EECS Department, University of Michigan, 1995.
130. S. Singh, *Fermat's Enigma: The Quest to Solve the World's Greatest Mathematical Problem*, Walker and Co., 1997.
131. A. P. Sistla, Safety, liveness and fairness in temporal logic, Formal Aspects of Computing 6(1994), 495–511.
132. A. P. Sistla, E.M. Clarke, Complexity of propositional temporal logics. Journal of the ACM, 32(1986), 733–749.
133. A. P. Sistla, E. M. Clarke, N. Francez, Y. Gurevich, Can message buffers be characterized in linear temporal logic, ACM Symposium on Principles of Distributed Computing 1982, Ottawa, Canada, 148–156.
134. A. P. Sistla, M. Y. Vardi, P. Wolper, The complementation problem for Büchi automata with applications to temporal logic, Theoretical Computer Science, 49(1987), 217–237.
135. N. Sharygina, D. Peled, A combined testing and verification approach for software reliability, Formal Methods Europe 2001, Lecture Notes in Computer Science 2021, Springer-Verlag, 611-628.
136. G. Stalmark, M. Säfinnd, Modeling and verifying systems and software in propositional logic, Safety of computer control systems (SAFECOMP'90), Pergamon Press, 1990, 31–36.
137. A. Szalas, L. Holenderski, Incompleteness of first-order temporal logic with Until. Theoretical Computer Science, 57(1988), 317-325.
138. A. S. Tanenbaum, *Computer Networks*, Prentice-Hall, Third edition 1996.
139. R. E. Tarjan, Depth first search and linear graph algorithms, SIAM Journal of computing, 1(1972).,146–160.
140. P. S. Thiagarajan, Elementary net systems, W. Brauer, W. Reisig, G. Rozenberg (eds.), Proceedings of Advances in Petri Nets 1986, Lecture Notes in Computer Science 254, Springer-Verlag, 1987, 26–59.
141. W. Thomas, Automata on infinite objects, In Handbook of Theoretical Computer Science, vol. B, J. van Leeuwen, ed., Elsevier, Amsterdam (1990) 133–191.
142. A. Valmari, A stubborn attack on state explosion, 2nd Workshop on Computer Aided Verification, New Brunswick, NJ, Lecture Notes in Computer Science 531, Springer-Verlag, 1990, 156–165.
143. M. P. Vasilevskii, Failure diagnosis of automata, Kibertetika, 4(1973), 98–108.
144. M. Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, Proceedings of the 1st Annual Symposium on Logic in Computer Science IEEE, 1986, 332–344.
145. S. Warshall, A theorem on Boolean matrices, Journal of the ACM, 9(1962), 11–12.
146. C. H. West, P. Zafiropulo, Automated validation of a communications protocol: the CCITT X.21 recommendation, IBM Journal of Research and Development 22(1978), 60–71.
147. P. Wolper, Temporal logic can be more expressive. Proceedings of the 22nd IEEE Symposium on Foundations of Computer Science, Nashville, TN, October 1981, 340–348.
148. P. Wolper, Expressing interesting properties of programs in propositional temporal logic, Principles of Programming Languages 1986, Florida, ACM Press, 184–193.
149. P. Wolper, V. Lovinfosse, Verifying properties of large sets of processes with network invariants. Proceedings of the 1989 International Workshop on Automatic Verification Methods for Finite State Systems, Lecture Notes in Computer Science 407, Springer-Verlag, 1989, 68–80.
150. P. Zafiropulo, A new approach to protocol validation, Proceedings of the international communications conference, Vol II (ICC 77), Chicago, 1977.
151. H. Zhang, A decision procedure for propositional logic, Association for Automated Reasoning Newsletter, 22, 1993, 1–3.

华章计算机科学丛书经典推荐

中/英第2版

深入理解计算机系统（原书第2版）
(美) Randal E. Bryant
David R. O'Hallaron 著
ISBN 978-7-111-32133-0
定价：99.00元

深入理解计算机系统（英文版第2版）
(美) Randal E. Bryant
David R. O'Hallaron 著
ISBN 978-7-111-32631-1
定价：128.00元



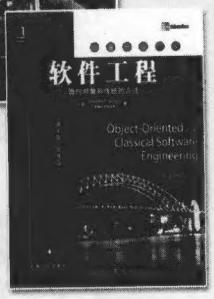
中/英第7版

软件工程：实践者的研究方法（原书第7版）
(美) Roger S. Pressma 著
ISBN 978-7-111-33581-8
定价：79.00元

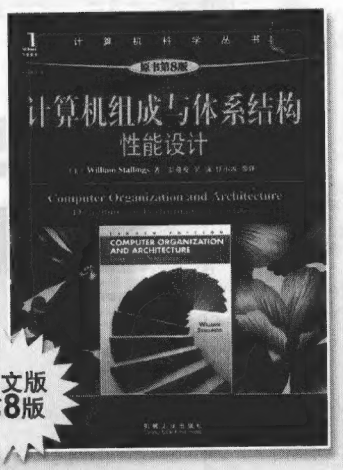
软件工程：实践者的研究方法（英文版第7版）
(美) Roger S. Pressma 著
ISBN 978-7-111-31871-2
定价：75.00元

中/英第8版

中文第8版即将出版

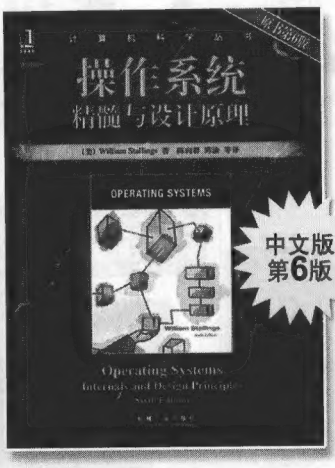


软件工程：面向对象和传统的方法（英文版第8版）
(美) Stephen R. Schach 著
ISBN 978-7-111-34196-3
定价：79.00元



中文版第8版

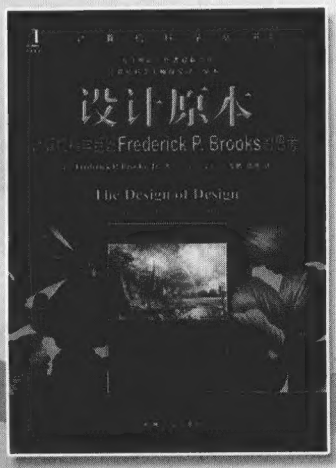
计算机组织与结构：性能设计（原书第8版）
(美) William Stallings 著
ISBN 978-7-111-32878-0
定价：79.00元



中文版第6版

操作系统：精髓与设计原理（原书第6版）
(美) William Stalling 著
ISBN 978-7-111-30426-5
定价：69.00元

- Stallings，大师级的人物，涉猎计算机安全、网络、体系结构等多方面，堪称计算机界的全才。
- 本书是其经典著作之一，得到全球计算机教育界和工程技术人员的好评！
- 以当代最流行的操作系统—Windows Vista、UNIX和Linux为例，全面清楚地展现了当代操作系统的本质和特点，具有先进性和适应性。



设计原本 ISBN 978-7-111-32557-4 定价：55.00

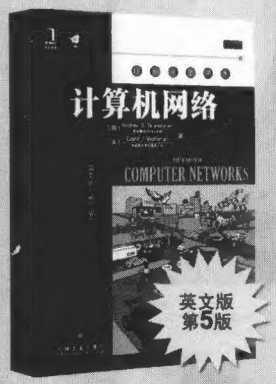
中文版第8版

Java语言程序设计 基础篇（原书第8版）
(美) Y. Daniel Liang 著
ISBN 978-7-111-34081-2
定价：75.00元

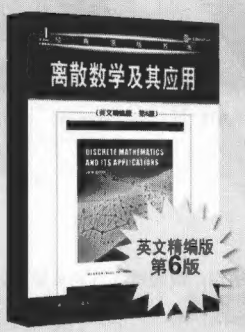
Java语言程序设计 进阶篇（原书第8版）
(美) Y. Daniel Liang 著
ISBN 978-7-111-34236-6
定价：79.00元

华章经典原版书库新版推荐

(美) Randal E. Bryant 著
David R. O'Hallaron
书号：978-7-111-32631-1
定价：128.00元

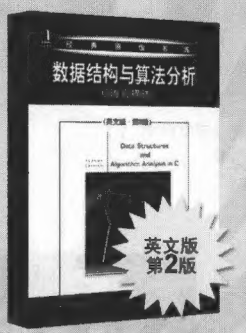


(荷) Andrew S. Tanenbaum 著
(美) David J. Wetherall
书号：978-7-111-35925-8
定价：99.00元



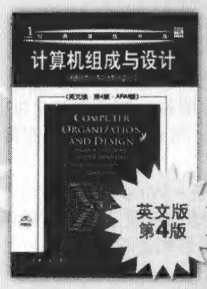
作者：Kenneth H. Rosen
书号：978-7-111-31329
定价：75.00

作者：Roger S. Pressman
书号：978-7-111-31871
定价：75.00



作者：Mark Allen Weiss
书号：978-7-111-31280
定价：45.00

作者：Alfred V. Aho 等
书号：978-7-111-32674-8
定价：78.00



作者：David A. Patterson 等
书号：978-7-111-30288-9
定价：95.00

作者：Bob Hughes 等
书号：978-7-111-30537-8
定价：39.00

作者：Stephen R. Schach
书号：978-7-111-34196-3
定价：85.00

作者：Umakishore Ramachandran 等
书号：978-7-111-31955-9
定价：69.00

作者：Ricardo Baeza-Yates 等
书号：978-7-111-33174-2
定价：78.00

作者：June Jamrich Parsons 等
书号：978-7-111-31799-9
定价：75.00

作者：Ramez Elmasri 等
书号：978-7-111-31094-5
定价：45.00

作者：M. Morris Mano 等
书号：978-7-111-30310-7
定价：58.00

作者：Kai Lai Chung
书号：978-7-111-30289-6
定价：49.00

作者：Pang-Ning Tan 等
书号：978-7-111-31670-1
定价：59.00

作者：Allan R. Hambley
书号：978-7-111-31459-2
定价：55.00

作者：Ian A. Glover
书号：978-7-111-31669-5
定价：98.00

教师服务登记表

尊敬的老师：

您好！感谢您购买我们出版的_____教材。

机械工业出版社华章公司为了进一步加强与高校教师的联系与沟通，更好地为高校教师服务，特制此表，请您填妥后发回给我们，我们将定期向您寄送华章公司最新的图书出版信息！感谢合作！

个人资料（请用正楷完整填写）

教师姓名		□先生 □女士	出生年月		职务		职称：□教授 □副教授 □讲师 □助教 □其他
学校			学院			系别	
联系电话	办公： 宅电： 移动：			联系地址及邮编			
				E-mail			
学历		毕业院校		国外进修及讲学经历			
研究领域							

主讲课程	现用教材名	作者及出版社	共同授课教师	教材满意度
课程： □专 □本 □研 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换
课程： □专 □本 □研 人数： 学期：□春□秋				□满意 □一般 □不满意 □希望更换

样书申请			
已出版著作		已出版译作	
是否愿意从事翻译/著作工作 □是 □否	方向		
意见和建议			

填妥后请选择以下任何一种方式将此表返回：（如方便请赐名片）

地 址：北京市西城区百万庄南街1号 华章公司营销中心 邮编：100037

电 话：(010) 68353079 88378995 传真：(010)68995260

E-mail:hzedu@hzbook.com marketing@hzbook.com 图书详情可登录http://www.hzbook.com网站查询